The development of semantic functions for a system description language with multiple interpretations

M. Seutter

The development of semantic functions for a system description language with multiple interpretations

Een wetenschappelijke proeve op het gebied
van de Wiskunde en Informatica.

Proefschrift
ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
maandag 31 januari 1994,
des namiddags te 1.30 uur precies
door

Marc Seutter

geboren op 29 september 1962 te De Bilt

Universitair Publicatiebureau KUN, Nijmegen

Promotor: Prof. dr. ir. R.T. Boute

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Seutter, Marc

The development of semantic functions for a system
description language with multiple interpretations / Marc Seutter
[S.l. : s.n.] (Nijmegen Universitair Publicatiebureau KUN).
Thesis Nijmegen.
ISBN 90-9006744-2
Subject headings: system semantics, computer science.

Acknowledgements

his thesis is a document that I intend writing only once. While writing I often wishe had never started it but having finished it I am glad the job is finally done. In thi eriod my relatives and friends have often suffered from my gloomy thoughts whethe would ever finish and I wish to thank all of them for their moral support.

My promotor and the members of the manuscript commission for reading m esis and giving much advice and many ideas.

Machiel van Frankenhuysen for reading and commenting chapter 2 of this thesis

My colleagues Erik, Mark, Niek, Kees, Franc, Paul, Ineke, Greta, Frank, Rie ien, Luc, Frans, John, Chris, Huub, Hans, Germaine, Veronique and Ineke.

The volunteers of the Technical Creative Center of Nijmegen Peter, Marcel, Roosj lav, Marlie, Frank and many others.

The members of Casus Belli and role playing Nijmegen Paul, Louis, Odile, Eril hn, Sjors, Dion-ben, Edgar, Maaike, Chantal, Jaap.

The members of Marie Curie.

Dear Rita for teaching me preciseness on the meaning of words and other thing

Frans of café De Fiets for his fluid support of many fruitful ideas.

My mother and brother.

Dear Marleen for your trust and love. I hope that our joy and bliss together wi ver end.

And especially dear Jolande for your love and belief that one time the job woul ally be finished. Our weekends together have lighted many dark places of the lon ad for which I am endlessly grateful.

Manuscriptcommissie:

Prof. dr. ir. M. Rem, Technische Universiteit Eindhoven
Prof. dr. ir. O.E. Herrmann, Universiteit Twente
Dr. ir. E.F.A. Deprettere, Technische Universiteit Delft
Prof. dr. ir. J.M. van Campenhout, Universiteit Gent

Contents

0 Introduction 1

1 Funmath 5

1.1 Introduction . . . 5
1.2 Basic principles . . . 6
1.2.1 Orthogonality . . . 6
1.2.2 Functions . . . 6
1.2.3 Identifiers . . . 7
1.2.4 Application . . . 7
1.2.5 Tuples . . . 7
1.2.6 Abstraction . . . 8
1.3 Types . . . 8
1.3.1 Definition . . . 8
1.3.2 The universal sets $\mathcal{T}$, $\mathcal{U}$ and $\mathcal{F}$ . . . 9
1.3.3 Other predefined types . . . 9
1.4 Transformational reasoning . . . 10
1.5 Functions and operators . . . 11
1.5.1 More on functions . . . 11
1.5.2 Operations associated with functions . . . 12
1.5.3 General-purpose higher order functions . . . 13
1.6 Cartesian products . . . 14
1.6.1 Definition . . . 14
1.6.2 Applied to an identifier . . . 15
1.6.3 Applied to an application . . . 15
1.6.4 Applied to a tuple . . . 15
1.6.5 Applied to an abstractor: dependent types . . . 16
1.6.6 Direct sums . . . 16
1.7 Genericity and polymorphism . . . 16
1.7.1 Overloading . . . 16
1.7.2 Polymorphism . . . 17
1.8 Sequences in Funmath . . . 17
1.9 Dyadic operators . . . 19
1.10 Local definitions . . . 21

1.11 Algebras in Funmath 2
1.12 Describing mathematics in context 2
1.13 Conclusions and final remarks 2

Using Funmath in mathematics **2**
2.1 Introduction 2
2.2 Boolean algebra 2
2.3 Group theory 2
2.4 Topology 3
2.4.1 Introduction 3
2.4.2 The definition of a topology 3
2.4.3 Bases of a topology 3
2.4.4 Topologies derived from a metric 3
2.4.5 Elementary topological notions 3
2.4.6 Continuous mappings 3
2.4.7 Conclusions 4

Informal Introduction to the language Glass **4**
3.1 Introduction 4
3.2 Systems semantics 4
3.2.1 Motivation 4
3.2.2 The principle of systems semantics 4
3.3 Basic language design 4
3.3.1 Terminology 4
3.3.2 Directionality of systems 4
3.3.3 Kernel and macro language 4
3.4 Directional systems 4
3.4.1 A very simple system 4
3.4.2 Multiple connections 4
3.4.3 Anonymous systems 4
3.4.4 Feedback and Fanout 4
3.4.5 Description of tristate interfaces 5
3.5 Some simple semantic functions 5
3.6 Adirectional and hybrid systems 5
3.6.1 Properly adirectional systems 5
3.6.2 Global connections in descriptions 5
3.6.3 Anonymous adirectional systems 5
3.6.4 Partially adirectional systems 5
3.6.5 Two illustrative hybrid examples 5
3.7 Typing and parametrization 5
3.7.1 Function types 5
3.7.2 Other primitive types 5
3.7.3 Parametrized atoms 5

3.7.4 Compound types . . . 58
3.7.5 Parametrized types . . . 59
3.7.6 Type naming . . . 59
3.7.7 Type variables . . . 59
3.8 Macros . . . 60
3.8.1 A first expansion . . . 60
3.8.2 Calculation in macros . . . 60
3.8.3 Using patterns . . . 61
3.8.4 A larger example . . . 62
3.8.5 A checkerboard example . . . 63
3.9 From Glass to Reals . . . 66

4 Simple semantic functions **67**
4.1 Introduction . . . 67
4.2 The generic model for the directional subset . . . 68
4.3 The simplex model . . . 69
4.4 Eichelberger algebra . . . 71
4.5 Semantic functions and tristate logic . . . 72
4.6 Some very simple cost models . . . 74
4.7 Conclusions . . . 75

5 Implementation aspects **77**
5.1 History of the Forfun environment . . . 77
5.2 The current describing environment . . . 78
5.3 The Forfun directory . . . 79
5.4 Using Tm . . . 80
5.5 Using Tm in C . . . 81
5.6 An example . . . 83
5.7 Final remarks . . . 84

6 Discrete timing function models **85**
6.1 Introduction . . . 85
6.2 The generic model revisited . . . 85
6.3 Direct extension . . . 87
6.4 The properties of the atoms . . . 88
6.5 Explicit clock signals . . . 91
6.6 First implementations . . . 92
6.7 An efficient implementation . . . 93
6.8 The actual implementation . . . 95
6.9 Final remarks . . . 95

7 Verification of hardware **97**
7.1 Introduction . . . 97

7.2 Natural number representation . . . 98
7.2.1 Representation and interpretation functions . . . 98
7.2.2 Binary addition . . . 101
7.2.3 Truncation . . . 103
7.2.4 Special case: derivation of an incrementer . . . 103
7.3 Towards sequential circuits . . . 105
7.3.1 The time model . . . 105
7.3.2 A first step in design . . . 107
7.3.3 Finding an appropriate subsystem . . . 110
7.3.4 The final derivation . . . 111
7.3.5 The complete Glass description . . . 113
7.4 Conclusions . . . 114

8 Considerations regarding asynchronous circuits 115
8.1 Introduction . . . 115
8.2 Race and hazard models . . . 116
8.2.1 Four representative models . . . 116
8.2.2 Ternary simulation . . . 116
8.3 Simulation of hardware . . . 118
8.3.1 First approximation: fixed delay model . . . 118
8.3.2 Models based on functionals . . . 120
8.3.3 Discrete event simulation . . . 121
8.4 Conclusions . . . 122

9 VHDL versus Glass 123
9.1 VHDL . . . 123
9.1.1 Introduction . . . 123
9.1.2 Entities . . . 123
9.1.3 Describing structure in VHDL . . . 125
9.1.4 The structural description of a data selector in VHDL . . . 126
9.1.5 Describing behaviour in VHDL . . . 127
9.1.6 The behavioural description of a data selector in VHDL . . . 128
9.2 VHDL and Glass . . . 128
9.3 VHDL contra Glass . . . 129
9.4 Conclusions . . . 129

10 Final remarks 131

A Kernel language datastructure definition file 133

B Implementation language equivalents 135
B.1 Equivalents for section 3.4 . . . 135
B.2 Equivalents for section 3.6 . . . 136

B.3 Equivalents for section 3.7 137
B.4 Equivalents for section 3.8 138

Chapter 0

Introduction

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with weary feet,
Until it joins some larger way
Where many paths and errands mee
And whither then? I cannot say.
Lord of the Rings *J.R.R. Tolkie*

urrent hardware design has become so complex that one needs formal techniques an uitable formalisms in describing hardware as well as its verification. When designin ardware a designer has to consider various aspects of the design such as its structur ehaviour, cost, layout, etc, which should be covered by the description formalisı sed. Current hardware description languages are quite deficient: oft one can onl escribe one aspect of the hardware in the language, whereas other aspects must b escribed in a different one. Or, several descriptions must be given in one language t escribe different aspects of the hardware. An example of such a language is VHDL.

A different approach is systems semantics [Bou88]. Its basic principle is the fa iat many physical systems and artifacts require a larger variety of different mode iews) than immaterial artifacts such as programs, and hence require additional a ention to notational economy in their description. One describes hardware formall sing an appropriate syntax and scoping rules based upon the Euler notation for fun ons. Only *one* language is used for the hardware description. Such a description do ot have an *a priori interpretation*: it merely describes formally the decomposition system into subsystems and their connectivity. Such a formal description may the e assigned a meaning (interpretation) by applying a *semantic function* to it. Sema c functions map a description onto a desired aspect of the hardware described. E pplying several semantic functions to *one* description one may obtain various aspec f the described system.

The primary aim of Esprit project 881, "Forfun" (started May 1986, ended May 1990), was a feasability study of the concept of systems semantics and (if possible) the development of an experimental prototype "describing environment". In the project a language, called Glass has been developed in which hardware may be described formally. A set of tools and semantic functions has been developed during the project. These semantic functions were programmed in usual programming languages (C, Pascal and Miranda). Therefore this thesis is also an engineering report on the Glass environment.

In this thesis we will describe the semantic functions in Funmath. Funmath is a language developed at this university for describing mathematical objects. It supports the description of functions over continuous and discrete domains pertaining to the theory of systems, electronics and programming. However, Funmath has a much wider scope: in this thesis we will also investigate how well traditional mathematics may be described in Funmath. The Funmath notation is very suitable for *transformational reasoning*, which will be used throughout this thesis as the preferred proof style.

Thus this thesis is centered around two languages: Glass and Funmath. Glass is used as a hardware description language and Funmath is used in its wider scope in describing mathematics and in its smaller scope in describing semantic functions. In one chapter we use Funmath as a metalanguage to reason about circuits. In some chapters we will also indicate how such a semantic function described in Funmath can be reformulated in existing programming languages.

Chapter 1 introduces the basic principles of Funmath. It also describes the preferred proof style of transformational reasoning. Several very useful (predefined) higher order functions are presented, together with (some of) their properties.

In the next chapter we investigate how well classical mathematical fields can be described in Funmath. We will treat three different mathematical fields namely boolean algebra, group theory and topology. We will not describe these fields in full detail but merely their roots, since the purpose of this chapter is to investigate the descriptional power of Funmath. As we will see, the functional notation of Funmath makes certain descriptions more concise and therefore easier to transform than the same descriptions in a classical mathematical notation.

The third chapter presents the motivation for our Esprit project and introduces the concepts of systems semantics. Next, the language Glass is described: its basics, its directional part, its adirectional part and its macro expansion mechanism.

The next chapter presents several simple semantic functions. A common generic model for these semantic functions is introduced. These semantic functions concern the static behaviour of digital circuits, a variant hereof based upon Eichelberger algebra, another variant handling tri-state logic. Some simple cost models are also discussed.

Chapter 5 presents a brief history of the project, the current describing environment and the problems encountered during its development. We discuss some solutions found for these problems. Lastly the support given by the environment for semantic function writers is presented, especially the support for the programming language C.

In chapter 6 two semantic functions are described concerning the dynamic behaviour of synchronous digital circuits, based on discrete time functions. We introduce

suitable notation which helps in manipulating these functions. An extension of the generic model introduced in chapter 4 is presented. A first implementation of this model in Miranda is discussed. Finally an efficient implementation of this model in C is presented.

In chapter 7 a hardware design approach is presented by which hardware is designed by formally proving its properties. One starts a design by formally specifying its high level abstract behaviour. This is then gradually transformed into more detailed low level characteristics. Finally a description is obtained that fits the properties of the interconnection of certain primitive components. We may see this as obtaining the inverse image under a semantic function, giving a (correct) Glass description of the specified circuit. This process is carried out for a combinational and a sequential circuit.

Chapter 8 presents some considerations and discussions concerning semantic functions concerning the dynamic behaviour of asynchronous digital circuits. Two models that have been implemented by the author are discussed.

In chapter 9 a comparison is made between the hardware description languages VHDL and Glass. Some concepts of VHDL are introduced. The advantages and disadvantages of both languages are discussed.

Some final remarks and conclusions are formulated in the last chapter.

Many people have influenced these various ideas through useful discussions. Specifically, however, the author's own contributions are the following:

- With respect to Funmath: the investigation of various usages in classical mathematics, such as topology, and its usage as a metalanguage for Glass.
- With respect to Glass: the language definition (in the context of Esprit project 881 – Forfun) and the implementarion of various parsers.
- With respect to semantic functions: the formal definition of most semantic functions and the implementation of all semantic functions described in this thesis.

Chapter 1

Funmath

1.1 Introduction

There was Eru, the One, who in Arda is called Iluvatar; and he made first the Ainur, the Holy Ones, that were the offspring of his thought; and they were with him before aught else was made.
Ainulindalë *J.R.R. Tolkien*

In principle, this chapter can be skipped by anyone familiar with traditional mathematical notation. Indeed, Funmath (*Fun*ctional *math*ematics) is meant as a "self-effacing" language: it is in the first place a restructuring and unification of the common notation in mathematics, providing surprising useful generalizations and new insights, and incidentally also justifying and systematizing traditional notational "shortcuts". The resulting generalisation justifies reading this chapter anyway.

In this thesis we will heavily use Funmath as a language for describing mathematical objects. Funmath is meant to describe functions over continuous and discrete domains pertaining to the theory of systems, electronics and programming. However, the scope turns out to be much wider. In a later chapter we will investigate how traditional mathematical fields, like topology and algebra, can be expressed in Funmath.

Clearly such a general language is not meant to be implemented in its entirety, as it allows to write nonexecutable, yet meaningful, mathematical expressions (e.g. equations). However two implementable subsets of Funmath are considered for special purposes:

- Comma (*Com*putational *ma*thematics), which is the executable subset of Funmath. As such it is a full-fledged functional programming language.

- Reals (*Real*izable *s*ystems), which constitutes the part of Funmath with which systems may be described. This language will be the successor of the system

description language Glass. The latter will be discussed in a later chapter of this thesis.

As is indicated by its name, most expressions in Funmath are mathematical functions or function applications. Functions have the advantage of having very useful manipulative properties, which supports transformational reasoning. They also have high expressive power, especially higher order functions. We will, however, be quite pragmatic in the sense that, if certain concepts can not be expressed conveniently as functions, nonfunctional entities will be introduced.

The notation is very suitable for transformational reasoning, which will be used throughout this thesis as the preferred proof style. Oft we will restrict ourselves to equational reasoning using term replacement as the only inference rule. Transformational proofs have the advantage of being linear, easy to read and understand. Although we prefer this style of reasoning, we will not restrict ourselves to purely transformational proofs. Moreover it is not always possible to find a convenient transformational proof for a certain theorem. In those cases we will fall back on a general deductive proofstyle if necessary.

1.2 Basic principles

1.2.1 Orthogonality

Funmath is based on only four *orthogonal* syntactic constructs:

- identifier
- application
- tuple
- abstraction

The first three support the variable-less style of description; the fourth supports the style with variables. The first two notions may denote any object; the latter two denote functions.

1.2.2 Functions

A *function* from a set X to a set Y is a mathematical entity that is characterized by:

- a domain, which is the set X
- a mapping, which associates with every element x in X a unique element in Y, called the *image* of x under f

The domain of a function f is denoted by $\mathcal{D}f$. If x is in $\mathcal{D}f$, we denote its image under f by $f\ x$. The *range* of f, written $\{f\}$ is the set of all images $f\ x$ for x in $\mathcal{D}f$. We say that f has *codomain* C, written as $f\ cod\ C$, if $\{f\} \subseteq C$. Clearly the codomain of a function need not be unique. The function with empty domain is unique and denoted by ε. The set of all functions from X to Y is written as $X \to Y$.

1.2.3 Identifiers

Identifiers denote objects by their names. An identifier is either a *constant* or a *variable* (part of an abstraction). Many constants are predefined such as the names of the basic types ($\mathbb{N}, \mathbb{Z}$, etc.), basic values ($0, 1, 42.37, 3i, \pi$, etc.), and basic operators ($+, \wedge$, etc.).

New constants are introduced by a *definition*. Such a definition defines a new mathematical entity (object) by specifying the type of the entity and by defining a characterizing trait of the entity by means of a defining proposition. The entity defined may also be a type. The syntax of such a definition is as follows [Bou92]:

def $x : X$ **with** P

This introduces a new constant x satisfying $x \in X \wedge P$. The writer must ensure that the entity is well-defined, that is: the defining proposition must be such that there is one and only one element in X that satisfies the proposition (proof obligation for the writer).

Examples:

def $y : \mathbb{N}$ **with** $y + 4 = 78$	— correct
def $z : \mathbb{C}$ **with** $z^2 = -1$	— incorrect, 2 solutions
def $k : \mathbb{Q}$ **with** $k^2 = 2$	— incorrect, 0 solutions
def $sqr : \mathbb{R} \to \mathbb{R}$ **with** $sqr\ x = x \cdot x$	— correct

1.2.4 Application

An application denotes an object which is the image of an (other) object under a suitable function. For example: $sqr\ 2$ denotes the number 4; $\mathcal{D}\ sqr$ denotes the set $\mathbb{R}$.

1.2.5 Tuples

A tuple denotes a function whose domain is a finite subset of $\mathbb{N}$. We will use the notation $\square n$ for the set containing the first n natural numbers. For example:

x, y, z

is a function with domain $\square 3$, such that $(x, y, z)\ 0 = x$, $(x, y, z)\ 1 = y$ and $(x, y, z)\ 2 = z$. Remark that the range of this function is $\{x, y, z\}$. A tuple consisting of a single element x is written as τx. The singleton set is written as $\iota x = \{\tau x\}$. In the same way twodimensional tuples may be used for denoting matrices.

1.2.6 Abstraction

In principle it is possible to do mathematics using only constants, applications and tuples. However the usage of variables has many advantages, recognized since their introduction by the greek mathematicians of the ancient world.

In Funmath variables are introduced by abstractions. An abstraction is a function denoted in the following way:

$x : X \mathbb{A} P \,.\, E$	(Rooijakkers variant [Roo93])
$E \mid x : X \mathbb{A} P$	(van Thienen variant [Thi93])

where x is a variable, or a tuple (recursively) composed of variables, X a type, P a filtering proposition and E an expression. Such an abstraction denotes a function whose domain is the collection of the elements of X that satisfy the filtering proposition P, such that the image of x equals E under this function. One may write

$$x : X \,.\, E \quad \text{for} \quad x : X \mathbb{A} 1 \,.\, E$$

Remark that this abstractor means the same as the λ abstractor occurring in typed λ calculus, with the difference that the superfluous λ is omitted and a filtering proposition may restrict the domain.

For the van Thienen variant the following abbreviations were introduced:

$E \mid x : X$	for	$E \mid x : X \mathbb{A} 1$
$x : X \mid P$	for	$x \mid x : X \mathbb{A} P$

Remark that these abbreviations combined with the range operator facilitates writing sets in the usual mathematical notation. Examples:

$x : \mathcal{D} f \mathbb{A} f\, x \in \mathcal{D} g \,.\, g\, (f\, x)$
$\{2 \cdot n \mid n : \mathbb{N}\}$
$\{m : \mathbb{N} \mid m < 5\}$
def $\square : \mathbb{N} \cup \iota\infty \to \mathcal{P}\mathbb{N}$ **with** $\square\, n = \{m : \mathbb{N} \mid m < n\}$
def $\blacksquare : \mathbb{N} \to \mathcal{P}\mathbb{N}$ **with** $\blacksquare\, n = \{m : \mathbb{N} \mid m \leq n\}$

1.3 Types

1.3.1 Definition

For what concerns the users view, a type is a set (or class) of mathematical entities, together with a set of operators over those entities (although this association is not enforced by the language). The reader may wonder why Funmath types are not limited to (proper) sets. The rationale behind this is it allows very elegant definitions of several useful operators. At this stage we will not be overly concerned with the exact definition of these "sets" in a way that avoids the known paradoxes. Several options are under consideration, one of them being stratification [For92].

For expository reasons, since it allows clear and compact descriptions of very general operators, we shall assume the existence of three "universal" sets:

- $\mathcal{U}$, the set of all mathematical objects
- $\mathcal{T}$, the set of all types
- $\mathcal{F}$, the set of all functions

Types are either predefined, corresponding to the sets one would normally expect such as $\mathbf{B}, \mathbf{Z}, \mathbf{N}, \mathbf{Q}, \mathbb{R}, \mathbf{C}, \mathcal{T}, \mathcal{U}, \mathcal{F}, \emptyset$ (on purpose we are not more precise on this issue), or defined by type expressions.

1.3.2 The universal sets $\mathcal{T}$, $\mathcal{U}$ and $\mathcal{F}$

The following operators are associated with the types $\mathcal{T}$, $\mathcal{U}$, and $\mathcal{F}$, occurring usually in type expressions:

operator	*example*	*explanation*
$\mathcal{P}$	$\mathcal{P}X$	the set of all subsets of X
$\setminus$	$X \setminus Y$	set difference
ι	ιx	the singleton set containing x
$\rightarrow$	$X \rightarrow Y$	type of all functions from X to Y
$\in$	$x \in X$	x is of type X
$\subseteq$	$X \subseteq Y$	X is a subset of Y

Remark that $\mathcal{T} \in \mathcal{T}$, $\mathcal{U} \in \mathcal{T}$ and $\mathcal{F} \in \mathcal{T}$.

1.3.3 Other predefined types

We define the set $\mathbf{B} = \{0, 1\}$, rather than $\{\mathbf{false}, \mathbf{true}\}$. This convention uncovers a large number of useful mathematical properties not shared by the traditional "isolated" set of truth values. By this convention we may write a conditional expression as $(a, b)\ c$, which may be read as **if** c **then** b **else** a **fi**. Although this notation is quite sufficient, we introduce the notation $c?a\natural b$ as syntactic sugar for $(b, a)\ c$ thus reducing the need for parenthesis as in $c?u\natural d?v\natural w$.

The predefined types occurring in arithmetic and of some of the associated operators are given by the following table:

Symbol	Name	Associated operators
$\emptyset$	empty set	none (or all)
$\mathbf{B}$	boolean	$\Rightarrow, \Leftrightarrow, =, \oplus, \vee, \wedge, <, >, \leq, \geq, \sqcap, \sqcup, +, -, \mathbf{mod}, \mathbf{div}, /, ., \neg$
$\mathbf{N}$	natural	$=, <, >, \leq, \geq, \sqcap, \sqcup, +, -, \mathbf{mod}, \mathbf{div}, /, \cdot$
$\mathbf{Z}$	integer	$=, <, >, \leq, \geq, \sqcap, \sqcup, +, -, \mathbf{mod}, \mathbf{div}, /, \cdot$
$\mathbf{Q}$	rational	$=, <, >, \leq, \geq, \sqcap, \sqcup, +, -, \mathbf{mod}, \mathbf{div}, /, \cdot$
$\mathbf{R}$	real	$=, <, >, \leq, \geq, \sqcap, \sqcup, +, -, \mathbf{mod}, \mathbf{div}, /, \cdot$
$\mathbf{C}$	complex	$=, +, -, /, \cdot$

In this table sets are listed in their usual (subset) order. The associated operators are listed in increasing order of precedence. Remark that we do not require that the sets are closed under the associated operator. As in usual mathematics some operators impose restrictions on the domains (2/0 is a meaningless expression). We will treat operators in more detail in a later paragraph.

As we may verify, this table contains some redundant operators: for instance $\wedge$ is the same as the multiplication operator restricted to $\mathbb{B}^2$. We still retain the typical syntax for logical operators for the following reasons:

- precedence (less parentheses)
- logical operators have useful algebraic properties that are not shared by the operators whose restrictions they are.

The arithmetic types may be extended with the values $-\infty$ or ∞ whenever this is mathematically meaningful.

1.4 Transformational reasoning

Transformational reasoning is more a *matter of style*, than a basic principle of the language. Whenever possible we will

- formulate axioms as equalities
- use substitution as main inference rule
- present a proof as a chain of proof steps of the form:

$$\begin{array}{ll} & E_0 \\ = \{justification_0\} & E_1 \\ = \{justification_1\} & E_2 \\ \quad \vdots & \\ = \{justification_{n-1}\} & E_n \end{array}$$

- formulate lengthy local derivations as lemmas

We will call a proof adhering to these rules an *equational transformational* proof. However not all proofs that we give can be formulated in an equational way. In general we will allow any relational operator, i.e. an operator denoting a function with codomain $\mathbb{B}$ between two proof steps. We will call a proof of the form

$$\begin{array}{ll} & E_0 \\ op_0\ \{justification_0\} & E_1 \\ op_1\ \{justification_1\} & E_2 \\ \quad \vdots & \\ op_{n-1}\ \{justification_{n-1}\} & E_n \end{array}$$

also a transformational proof if it adheres to the other rules given above. Its meaning is then:

$$(E_0\ op_0\ E_1) \wedge (E_1\ op_1\ E_2) \wedge \ldots \wedge (E_{n-1}\ op_{n-1}\ E_n).$$

For instance if op_i is $<$ and the other op_k $(k \neq i)$ are $=$ or $\leq$, then $E_0 < E_n$.

1.5 Functions and operators

1.5.1 More on functions

We will call the defining proposition in a function definition an *image definition*. This image definition may either be *intensional*, that is written as a formula, or it may be *extensional*, that is expressed by a table. Free variables occuring in an image definition are implicitly universally quantified over the domain of the function.

Two functions are *equal* if they have the same domain and the same mapping. Formally:

$$f = g \Leftrightarrow \mathcal{D}f = \mathcal{D}g \wedge \forall(x : \mathcal{D}f \cdot f\ x = g\ x)$$

We will call a function a *higher order function* if its domain and/or its codomain is a function type. For example:

def $times : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$
with $(times\ x)\ y = x \cdot y$

As notational convention we will mean by $A \to B \to C$ the type $A \to (B \to C)$ and by $f\ a\ b$ we will mean $(f\ a)\ b$.

Functions will often be used as operators in expressions for reasons of simplicity, clarity or compatibility with usual mathematical conventions. We introduce extra syntax to indicate whether a function is used as a prefix, infix, postfix, subscripted or even multifix operator. We indicate the location of operands in an application of the function to be defined by indicating these locations in the type declaration of the function by dashes (—). For example:

def $—! : \mathbb{N} \to \mathbb{N}$
with $a! = (a = 0)?1\dagger a \cdot (a - 1)!$

This specification of operand locations may be omitted if the function to be defined will be used as a prefix operator.

There are no syntactical constructs in Funmath yet to specify the associativeness and priority of user defined operators, apart from the general rule that monadic operators always have higher precedence than polyadic operators. In a future version of the language these constructs will be incorporated in the language definition.

1.5.2 Operations associated with functions

We introduce the following operators:

- the *bijective domain*

  def $\mathcal{B} : \mathcal{F} \to \mathcal{T}$
  with $\mathcal{B}\, f = \{x : \mathcal{D}f \mid \forall(y : \mathcal{D}f \cdot f\, x = f\, y \Leftrightarrow x = y)\}$

- *injectiveness*

  def $\mathit{—is\ injective} : \mathcal{F} \to \mathbf{B}$
  with $f\ \mathit{is\ injective} = (\mathcal{B}\, f = \mathcal{D}f)$

- *surjectiveness*

  def $\mathit{—is\ surjective\ over—} : \mathcal{F} \to \mathcal{T} \to \mathbb{B}$
  with $f\ \mathit{is\ surjective\ over}\ A = \{f\} = A$

- *bijectiveness*

  def $\mathit{—is\ bijective\ over—} : \mathcal{F} \to \mathcal{T} \to \mathbf{B}$
  with $f\ \mathit{is\ bijective\ over}\ A = f\ \mathit{is\ injective} \wedge f\ \mathit{is\ surjective\ over}\ A$

- the *bijective range.*

  def $\mathcal{R} : \mathcal{F} \to \mathcal{T}$
  with $\mathcal{R}\, f = \{x : \mathcal{B}\, f \cdot f\, x\}$

  Recall that the normal range of a function f is indicated by $\{f\}$.

- the *inverse function*

  def $\overline{\ —\ } : \mathcal{F} \to \mathcal{F}$
  with $f^{-} \in \mathcal{R}\, f \to \mathcal{B}\, f \wedge f\, (f^{-}\, x) = x$

  The introduction of $\mathcal{B}$ and $\mathcal{R}$ allows to define and use inverses in a very general way without explicitly determining $\mathcal{B}$ and $\mathcal{R}$ in a particular case. This offers the possibility to leave certain items unspecified until the need for a precise specification arises. Another example is:

  $$A \rightarrowtail B = \{f : \mathcal{F} \mid \mathcal{D}f \subseteq A \wedge f\ \mathit{cod}\ B\}$$

  which replaces the concept of partial functions.

- the *constant function specifier*

$$\textbf{def}\ \ —^{\bullet}— : \mathcal{U} \times \mathcal{T} \to \mathcal{F}$$
$$\textbf{with}\ a^{\bullet}A \in A \to \iota\, a$$

- the *domain restrictor*

$$\textbf{def}\ \ —\rceil— : \mathcal{F} \times \mathcal{T} \to \mathcal{F}$$
$$\textbf{with}\ f\rceil A \in \{x : \mathcal{D}f \mid x \in A\} \to \{f\} \wedge \forall(x : \mathcal{D}f \mathbb{A} x \in A \,.\, (f\rceil A)\, x = f\, x)$$

- the *codomain specifier*

$$\textbf{def}\ \ Fcod : \mathcal{T} \to \mathcal{T}$$
$$\textbf{with}\ Fcod\ A = \{f : \mathcal{F} \mid f\ cod\ A\}$$

- *function composition*

$$\textbf{def}\ \ —\circ— : \mathcal{F} \times \mathcal{F} \to \mathcal{F}$$
$$\textbf{with}\ g \circ f = (x : \mathcal{D}f \mathbb{A} f\, x \in \mathcal{D}g \,.\, g\,(f\, x))$$

Remark that $\{f\} \cap \mathcal{D}g = \emptyset \Leftrightarrow g \circ f = \varepsilon$
Remark also $\{f\} \subseteq \mathcal{D}g \Leftrightarrow \forall(x : \mathcal{D}f \,.\, f\, x \in \mathcal{D}g) \Leftrightarrow \mathcal{D}(g \circ f) = \mathcal{D}f$

- *autocomposition* or n-fold composition of f with itself

$$\textbf{def}\ \ —^{\frown} : \mathcal{F} \to \mathbf{Z} \to \mathcal{F}$$
$$\textbf{with}\ f^n = (n = 0)?id_{\mathcal{D}f}\dagger(n > 0)?f \circ f^{n-1}\dagger f^{\frown} \circ f^{n+1}$$

The reader may verify that if f is injective the following properties hold:

$$\forall(m, n : \mathbf{Z}^2 \,.\, f^{m+n} = f^m \circ f^n)$$
$$\forall(m, n : \mathbf{Z}^2 \,.\, (f^m)^n = f^{mn})$$

1.5.3 General-purpose higher order functions

Since $A^2 = \square 2 \to A$ standard operators on A are already higher order functions in *Fcod A*. We extend them to a "larger" subset in *Fcod A* by defining a function F for a binary operator f in such a way that $F\,(a, b) = a\, f\, b$.

The following table introduces some often used higher order general-purpose operators, thus defined:

restr. *domain*	restr. codomain	*op* f	result type	meaning
—	—	$\{f\}$	$\mathcal{T}$	range of f
—	—	$\square f$	$\{f\}$	any element of $\{f\}$
—	$\mathcal{T}$	$\bigtimes f$	$\mathcal{T}$	generalized cartesian product
—	$\mathcal{T}$	$\bigcup f$	$\mathcal{T}$	$\{x : \mathcal{U} \mid \exists(X : \{f\} . x \in X)\}$
—	$\mathcal{T}$	$\bigcap f$	$\mathcal{T}$	$\{x : \mathcal{U} \mid \forall(X : \{f\} . x \in X)\}$
—	$\mathbb{B}$	$\forall f$	$\mathbb{B}$	$\neg(0 \in \{f\})$
—	$\mathbb{B}$	$\exists f$	$\mathbb{B}$	$1 \in \{f\}$
finite	$\mathbb{C}$	$\sum f$	$\mathbb{C}$	sum of images $f\ x$ as x ranges over $\mathcal{D}f$
finite	$\mathbb{C}$	$\prod f$	$\mathbb{C}$	product of images $f\ x$ as x ranges over $\mathcal{D}f$

If we apply these operators to ε, we obtain the following results:

$$\{\varepsilon\} = \emptyset \qquad \bigcap \varepsilon = \mathcal{U} \qquad \bigcup \varepsilon = \emptyset$$
$$\forall \varepsilon = \neg(0 \in \{\varepsilon\}) = 1 \qquad \exists \varepsilon = 1 \in \{\varepsilon\} = 0$$

Remark also that

$$\forall\forall = \neg(0 \in \{\forall\}) = \neg(0 \in \{0,1\}) = 0$$
$$\exists\exists = 1 \in \{\forall\} = 1$$

Some of these higher order operators are clear extensions of usual binary operators:

$$x \wedge y = \forall(x, y)$$
$$x \vee y = \exists(x, y)$$
$$x + y = \sum(x, y)$$
$$x \cdot y = \prod(x, y)$$

1.6 Cartesian products

1.6.1 Definition

The operator $\bigtimes$ is a generalization of the usual cartesian product:

def $\bigtimes : Fcod\ \mathcal{T} \to \mathcal{T}$
with $\bigtimes F = \{f : \mathcal{D}F \to \bigcup F \mid \forall(i : \mathcal{D}f . f\ i \in F\ i)\}$

In Funmath, cartesian products are function spaces, thus generalizing the notion of tuples. There is an analogy with function equality in the following way:

$$f = g \Leftrightarrow \mathcal{D}f = \mathcal{D}g \wedge \forall(x : \mathcal{D}f . f\ x = g\ x)$$
$$f \in \bigtimes g \Leftrightarrow \mathcal{D}f = \mathcal{D}g \wedge \forall(x : \mathcal{D}f . f\ x \in g\ x)$$

As a demonstration of the orthogonality of Funmath, we demonstrate how this higher order combinator combines with the four basic constructs.

1.6.2 Applied to an identifier

Consider the previously introduced function ■. Then:

$$\begin{array}{ll} & \bigtimes ■ \\ = \{\text{def.} \bigtimes\} & \{f : \mathcal{D}■ \to \bigcup ■ \mid \forall(i : \mathcal{D}■ \,.\, f\, i \in ■ i)\} \\ = \{\text{def.} ■\} & \{f : \mathbb{N} \to \mathbb{N} \mid \forall(i : \mathbb{N} \,.\, f\, i \in \{m : \mathbb{N} \mid m \leq i\})\} \\ = \{\text{logic}\} & \{f : \mathbb{N} \to \mathbb{N} \mid \forall(i : \mathbb{N} \,.\, f\, i \leq i)\} \end{array}$$

1.6.3 Applied to an application

Consider the application $B^{\bullet}A$. Remark that $\mathcal{D}(B^{\bullet}A) = A$ and that $\bigcup(B^{\bullet}A) = B$. Then:

$$\begin{array}{ll} & \bigtimes (B^{\bullet}A) \\ = \{\text{def} \bigtimes\} & \{f : \mathcal{D}(B^{\bullet}A) \to \bigcup(B^{\bullet}A) \mid \forall(i : \mathcal{D}(B^{\bullet}A) \,.\, f\, i \in (B^{\bullet}A)\, i)\} \\ = \{\text{def } {}^{\bullet}\} & \{f : A \to B \mid \forall(i : A \,.\, f\, i \in B)\} \\ = \{\text{def } \to\} & A \to B \end{array}$$

1.6.4 Applied to a tuple

Consider the tuple (A, B) with $A, B \in \mathcal{T}^2$. Then $\mathcal{D}(A, B) = \Box 2$, $(A, B)\ 0 = A$ and $(A, B)\ 1 = B$. $\bigtimes(A, B)$ is therefore the set of tuples whose first element is in A and whose second element is in B. We can therefore define the meaning of the usual Cartesian product:

$$A \times B = \bigtimes(A, B)$$
$$A \times B \times C = \bigtimes(A, B, C)$$

Likewise we can define:

$$A \cup B = \bigcup(A, B)$$
$$A \cap B = \bigcap(A, B)$$

Let A be a function of type $\Box n \to \mathcal{T}$ (In mathematics A is often called a *family* of sets and denoted in a redundant way by $i : \Box n \,.\, A\, i$). Clearly:

$$\bigtimes A = A\, 0 \times A\, 1 \times \ldots \times A\, (n-1)$$

or equivalently

$$\bigtimes A = \bigtimes(i : \Box n \,.\, A\, i)$$

If $f \in A \to B \to C$, $g \in A \times B \to C$ and $\forall(a, b : A \times B \,.\, f\, a\, b = g\, (a, b))$ then f is called the *Curried* version of g and g is called the *Cartesian* version of f.

1.6.5 Applied to an abstractor: dependent types

Consider the following function $F = a : A \mathbf{.} B$, where B depends on a. Then $\mathcal{D}F = A$ and $\bigcup F = \bigcup(a : A \mathbf{.} B)$. Then:

$$\bigtimes(a : A \mathbf{.} B) = \{f : A \to \bigcup(a : A \mathbf{.} B) \mid \forall(a : A \mathbf{.} f\, a \in B)\}$$

As we see, $\bigtimes(a : A \mathbf{.} B)$ stands for the usual dependent (function) type. We will abbreviate it by $A \ni a \to B$. An example of this construct is the following definition:

def $Limit : \mathbb{N} \ni n \to \square 2^{n+1} \to \square 2^n$
with $Limit\ n\ m = m$ **mod** 2^n

1.6.6 Direct sums

In analogy to the Cartesian product operator we can define the following higher order operator, which is a generalization of the direct sum:

def $+ : Fcod\ \mathcal{T} \to \mathcal{T}$
with $+F = \{f : \mathcal{D}F \times \bigcup F \mid f\ 1 \in F\ (f\ 0)\}$

Consider for instance:

$$+\,(A, A, A) = \square 3 \times A$$

In mathematics this construct is often called the disjoint union or sometimes labeled union. Consider also:

$$\begin{array}{ll} & +\blacksquare \\ = \{\text{def. } +\} & \{f : \mathcal{D}\blacksquare \times \bigcup\blacksquare \mid f\ 1 \in \blacksquare(f\ 0)\} \\ = \{\text{def. } \blacksquare\} & \{f : \mathbb{N} \times \mathbb{N} \mid f\ 1 \in \{m : \mathbb{N} \mid m \leq f\ 0\}\} \\ = \{\text{logic}\} & \{f : \mathbb{N} \times \mathbb{N} \mid f\ 1 \leq f\ 0\} \end{array}$$

The application of this operator to an abstraction introduces a dependent Cartesian product $+(a : A \mathbf{.} B)$, which we may abbreviate by $A \ni a \times B$.

1.7 Genericity and polymorphism

1.7.1 Overloading

As usual in mathematics, some operators are overloaded. Some provisions are taken in Funmath to do this in an orderly way, avoiding ambiguities. When overloading a symbol the writer must ensure that every application of that name is semantically unambiguous, that is its corresponding definition must be uniquely deducible from the context. There are two language supported cases of overloading, namely for overloading operators with their Curried versions and for polymorphism.

Operators that may be overloaded with their Curried versions are introduced by writing $\leftrightarrow$ between the operand types in the type declaration. The definition

def $\quad —op— : A \leftrightarrow B \to C$
with $a\ op\ b = expression$

indicates that *op* is threefold overloaded by:

def $—op— : A \times B \to C$ **with** $a\ op\ b = expression$
def $—op : A \to B \to C$ **with** $(a\ op)\ b = expression$
def $op— : B \to A \to C$ **with** $(op\ b)\ a = expression$

By convention all commonly used infix operators are assumed overloaded in this fashion.

1.7.2 Polymorphism

Polymorphism (Greek: $\pi o \lambda \upsilon$ = many, $\mu o \rho \phi \eta$ = form) is the usage of a single function symbol to designate a family of operators, each differing in the type of its operands but having formally identical image definitions. Polymorphic definitions can be introduced in two ways in Funmath, namely with or without overloading the function symbol.

The first way consists of adding the type as extra information using dependent types in the type expression. As example

def $\quad twice\_ : \mathcal{T} \ni A \to (A \to A) \to (A \to A)$
with $twice_A\ f = f \circ f$

This way of introducing polymorphism has the disadvantage that in every application of the function the type parameters must be given explicitly.

The other way of introducing polymorphic definitions uses overloading. A general and elegant approach to this kind of polymorphism is being elaborated by F. van den Beuken. Temporarily we use the following syntax, by which an implicit dependent type is bound by the keyword **poly**. For example:

poly $A : \mathcal{T}$ **.** **def** $\quad twice : (A \to A) \to (A \to A)$
with $twice\ f\ x = f\ (f\ x)$

Of course, the writer must ensure that the meaning of such a function symbol is determinable from the context.

1.8 Sequences in Funmath

In this section we will consider special cases of the generalized cartesian product. We introduce the following definition:

$$\textbf{def}\ \ - \uparrow - : \mathcal{T} \leftrightarrow \mathbf{N} \cup \iota\infty \to \mathcal{T}$$
$$\textbf{with}\ A \uparrow n = \bigtimes(A^{\bullet} \square n)$$

Since $\bigtimes(B^{\bullet}A) = A \to B$, we have that $A \uparrow n = \square n \to A$. We will abbreviate $A \uparrow n$ by A^n. We will call these types *array* types. The reader should be aware of the fact that by this definition $A^1 \neq A$. To map elements from a set A into A^1 we use the following (polymorphic) injection function:

$$\textbf{poly}\ A : \mathcal{T} \bullet\ \textbf{def}\ \ \tau : A \to A^1$$
$$\textbf{with}\ \tau\ a\ 0 = a$$

Clearly

$$\begin{array}{ll} & A^0 \\ = \{\text{def } \uparrow\} & \bigtimes(A \bullet \square 0) \\ = \{\text{prop } {}^{\bullet}\} & \square 0 \to A \\ = \{\square 0 = \emptyset\} & \emptyset \to A \end{array}$$

Hence A^0 equals the set containing the unique function whose domain is the empty set, denoted by ε. Since $\square\infty = \mathbf{N}$, $A^{\infty} = \mathbf{N} \to A$. We will call a type of this form a *stream* type. The following definitions introduce the so-called *sequence* types:

$$\textbf{def}\ \ -^{*} : \mathcal{T} \to \mathcal{T}$$
$$\textbf{with}\ A^{*} = \bigcup(n : \mathbf{N} \bullet A^n)$$

$$\textbf{def}\ \ -^{\omega} : \mathcal{T} \to \mathcal{T}$$
$$\textbf{with}\ A^{\omega} = A^{*} \cup A^{\infty}$$

Note that $A^{*} \cap A^{\infty} = \emptyset$. The following operators are defined on sequence types:

- the *length* operator:

$$\textbf{poly}\ A : \mathcal{T} \bullet\ \textbf{def}\ \ \# : A^{\omega} \to \mathbf{N} \cup \iota\infty$$
$$\textbf{with}\ \forall(x : A^{\omega} \bullet x \in A^{\#x})$$

- the *concatenation* operator

$$\textbf{poly}\ A : \mathcal{T} \bullet\ \textbf{def}\ \ +\!\!+ : A^{\omega} \ni x \leftrightarrow A^{\omega} \ni y \to A^{\#x+\#y}$$
$$\textbf{with}\ (x+\!\!+y)\ i = (i < \#x)?x\ i\dagger y\ (i - \#x)$$

- the *prefix* operator

$$\textbf{poly}\ A : \mathcal{T} \bullet\ \textbf{def}\ \ \succ : A \leftrightarrow A^{\omega} \ni x \to A^{\#x+1}$$
$$\textbf{with}\ a \succ x = \tau a +\!\!+ x$$

- the *postfix* operator

$$\textbf{poly } A : \mathcal{T} \textbf{ . def } \prec : A^\omega \ni x \leftrightarrow A \to A^{\#x+1}$$
$$\textbf{with } x \prec a = x +\!\!+ \tau a$$

- the *head* operator

$$\textbf{poly } A : \mathcal{T} \textbf{ . def } \alpha : A^\omega - A^0 \to A$$
$$\textbf{with } \alpha\ x = x\ 0$$

- the *tail* operator

$$\textbf{poly } A : \mathcal{T} \textbf{ . def } \sigma : A^\omega - A^0 \ni x \to A^{\#x-1}$$
$$\textbf{with } \sigma\ x\ n = x\ (n+1)$$

The following equalities hold:

$$\#\varepsilon = 0$$
$$\#\tau a = 1$$
$$\#(x +\!\!+ y) = \#x + \#y$$
$$\#a \succ x = 1 + \#x$$
$$\#x \prec a = 1 + \#x$$
$$(a \succ x)\ 0 = a$$
$$(a \succ x)\ (i+1) = x\ i$$
$$\alpha\ x \succ \sigma\ x = x$$
$$\sigma\ (a \succ x) = x$$

whose proof is left to the reader.

In the final version of Funmath, operators such as $+\!\!+$ are defined over heterogeneous tuples as well, with the property that:

$$x \in \bigtimes A \wedge y \in \bigtimes B \Rightarrow x +\!\!+ y \in \bigtimes (A +\!\!+ B)$$

1.9 Dyadic operators

All (usual) dyadic operators are overloaded with their Curried versions. For example:

$$\cdot : \mathbf{C} \leftrightarrow \mathbf{C} \to \mathbf{C}$$
$$(\cdot 3)\ (3 + 2i) = 9 + 6i$$

Many dyadic operators are also quite overloaded. We can group them as follows:

- Equality and Inequality

$$- = - : A \leftrightarrow A \to B$$
$$- \neq - : A \leftrightarrow A \to B$$

  are overloaded in the extreme: for every type A there exists a different definition. For the arithmetic domains they are viewed as operators over $\mathbf{C}$.

- Operators for partially ordered sets
  For many sets a partial order may be defined usually denoted by

  $$— < — : A \leftrightarrow A \to \mathbb{B},$$
  $$— \leq — : A \leftrightarrow A \to \mathbb{B}, \text{ etc.}$$

  This partial order often gives rise to the following higher order functions, provided A is a complete lattice:

  def $\bigsqcup : \mathit{Fcod}\ A \to A$
  with $\forall(a : A \bullet \bigsqcup f \leq a \Leftrightarrow \forall(x : \{f\} \bullet x \leq a))$

  def $\bigsqcap : \mathit{Fcod}\ A \to A$
  with $\forall(a : A \bullet a \leq \bigsqcap f \Leftrightarrow \forall(x : \{f\} \bullet a \leq x))$

  The first of these operators is often called the *join* operator. The other is called the *meet* operator. In the case of the arithmetic domains, these relational operators are operators over $\mathbf{R}$. The join operator $\bigsqcup$ then corresponds to *supremum* and the meet operator $\bigsqcap$ to *infimum*. Remark also that the dyadic counterparts of these operators correspond with *max* and *min*:

  $$3 \sqcup 7 = \bigsqcup(3, 7) = 7$$

- Arithmetic operators
  $+, -, \cdot, /, \uparrow$ are operators on $\mathbf{C}$. $\mathbf{div}, \mathbf{mod}, \sqcup, \sqcap$ are operators on $\mathbb{R}$.

- Logical operators
  Observe that we view many logical operators as restrictions of arithmetic or relational operators to $\mathbb{B}^2$. For instance

  $\Rightarrow$ is the restriction of $\leq$
  $\wedge$ is the restriction of $\sqcap$, etc.

  The same holds for the corresponding higher order functions: $\forall$ is the restriction of $\bigsqcap$ and $\exists$ is the restriction of $\bigsqcup$ to $\mathbb{B}$. However we will continue to denote the operators $\Rightarrow, \Leftrightarrow, \wedge, \vee, \oplus$, by their usual symbols because of their algebraic properties and their correspondence to existing notations.

For expressions in Funmath the following precedence rules hold:

- Monadic operators have precedence over infix operators.
- The usual precedence rules for arithmetic operators hold.
- Arithmetic operators have precedence over relational operators.
- Relational operators have precedence over logical operators.

- Among the logical operators the usual precedence rules hold.
- An abstractor has lowest precedence.
- Parentheses are used to change precedences.

1.10 Local definitions

An expression may contain a where construct, which serves to introduce local definitions. Its syntax is as follows:

$$E \textbf{ where } x : X \textbf{ with } P$$

which is syntactic sugar for the value

$$[](x : X \mathbb{A} P \centerdot E)$$

If the defining proposition is of the form $x = E'$ the type declaration may be omitted in the case that the type of x is uniquely deducible from the context. The scope of these local definitions is limited to the surrounding expression.

1.11 Algebras in Funmath

An *algebra* is a (possibly singleton) collection of unspecified nonempty sets together with a collection of unspecified operators satisfying:

- a collection of closure axioms
- a collection of axioms characterizing the specific algebra

An algebra (or more general a type with a certain mathematical structure) is described in Funmath as a predicate on the sets and their operators. A well known example is the following description of a *monoid*. Remark that the closure axioms are implicit in the type definition:

$$\begin{array}{l}\textbf{def} \quad \mathit{monoid} : \mathcal{T} \ni G \times (G \times G \to G) \to \mathbb{B} \\ \textbf{with } \mathit{monoid}\ (G, \cdot) = \forall (a, b, c : G^3 \centerdot a \cdot (b \cdot c) = (a \cdot b) \cdot c)\end{array}$$

Other examples of describing mathematical structures in Funmath can be found in the next chapter.

1.12 Describing mathematics in context

Many mathematical theories start by adding extra structure to a certain set in the form of axioms and by proving theorems and propositions on the basis of the added

extra structure. The definition of a group is an example of this. Many mathematical textbooks proceed after defining the necessary axioms by defining properties, functions and so on, which are only valid within the context of the axioms. We may think for instance of the definition of the cyclic subgroup of a group generated by one of its elements. The context in these definitions is then often described in natural language.

This approach, although frequently used in mathematics, can not be used in Funmath. In a context where many structures coexist, a more explicit way is necessary. Of course one could add the context as an extra parameter in every definition requiring much unnecessary writing. We therefore propose the following syntax for grouping a context of axioms and definitions together, which avoids explicit parameters (this syntax is not yet fixed in the language definition). It is somewhat similar to the package construct of Ada. By

packet *packetname*
type declaration **&** *filtering proposition* **.**

⋮

definitions, theorems, propositions and proofs

⋮

end packet

is meant that the definitions, theorems, and so on are only valid in the context of the sets declared in the type declaration that satisfy the filtering proposition (which may be used of course to impose the extra structure to the sets). If we wish to use these nested definitions elsewhere, we write:

refer *packetname*

followed by an instance of the sets declared in the packet heading. Examples of this syntax are used in the next chapter.

1.13 Conclusions and final remarks

This chapter introduces the language Funmath as a language to describe mathematical objects. Its design is very orthogonal, based upon only 4 basic constructs:

- constants (and syntax to introduce new ones)
- tuples
- function application
- abstraction

The language is strongly typed while providing support for polymorfic definitions.

Its design aims the description of functions over various domains pertaining t
ıgineering sciences such as the theory of systems, electronics and programming. Y
s scope is much wider. In the next chapter we will investigate how well Funmat
ıd the preferred proof style of transformational reasoning can be used in describin
assical mathematical fields. In chapters 4, 6, 7 and 8 we will investigate the usage
unmath in hardware description.

Chapter 2

Using Funmath in mathematics

From darkness to darkness / My voice echoes in emptiness.
From this world to the next / My voice cries with life.
From darkness to darkness I shout / Beneath my feet all is made firm.
Time that flows / Hold in your course.
Because by fate even the gods are cast down / Weep ye all with me.
War of the twins — *M. Weis and T. Hickman*

2.1 Introduction

In the following chapter we will investigate how well classical mathematical fields may be described in Funmath, how well our preferred proof style of transformational reasoning can be used in proving theorems and reasoning and how well the Funmath notation corresponds to the usual "standard" mathematical notation.

We will treat three fields in mathematics, namely boolean algebra, group theory and topology. It is not our intention to describe these fields in full, (which would require several volumes filled with mathematical theory) but merely to get a feeling of the usability of Funmath in mathematics. We will therefore restrict ourselves and discuss only very basic notions and proofs of these three fields.

2.2 Boolean algebra

Before we can formulate the Huntington axioms[Hun04] we will define some auxiliary functions:

def $commutes : \mathcal{T} \ni B \times (B \times B \to B) \to \mathbf{B}$
with $commutes\ (B, *) = \forall(a, b : B^2 \cdot a * b = b * a)$

def $distributes : \mathcal{T} \ni B \times (B \times B \to B) \times (B \times B \to B) \to \mathbf{B}$
with $distributes\ (B, *, \diamond) = \forall(a, b, c : B^3 \cdot a * (b \diamond c) = (a * b) \diamond (a * c))$

def *has unit* $\cdot\ \mathcal{T} \ni B \times (B \times B \to B) \times B \to \mathbb{B}$
with *has unit* $(B, *, e) = \forall(a : B \,.\, a * e = a \wedge e * a = a)$

def *has inverse to* $: \mathcal{T} \ni B \times (B \times B \to B) \times B \times B \times B \to \mathbb{B}$
with *has inverse to* $(B, *, e, a, a\prime) = (a * a\prime = e)$

def *Boolean algebra with unities* :
$\mathcal{T} \ni B \times (B \times B \to B) \times B \times (B \times B \to B) \times B \to \mathbb{B}$
with *Boolean algebra with unities* $(B, +, e_+, ., e) =$

commutates $(B, +) \wedge$ *commutates* $(B, .) \wedge$	--- $(C+, C.)$
distributes $(B, +, .) \wedge$ *distributes* $(B, ., +) \wedge$	--- $(D+, D.)$
has unit $(B, +, e_+) \wedge$ *has unit* $(B, ., e) \wedge$	--- $(E+, E.)$
$\forall(a : B \,.\, \exists(a\prime : B \,.$ *has inverse to* $(B, +, e, a, a\prime) \wedge$	--- $(I+)$
has inverse to $(B, ., e_+, a, a\prime)))$	--- $(I.)$

def *Boolean algebra* $: \mathcal{T} \ni B \times (B \times B \to B) \times (B \times B \to B) \to \mathbb{B}$
with *Boolean algebra* $(B, +, .) =$
$\exists(e_+, e : B \times B \,.$ *Boolean algebra with unities* $(B, +, e_+, ., e))$

Since we are going to prove several properties of boolean algebras we encapsulate these propositions and proofs in a packet:

packet *Basic Boolean Algebra*
$B, +, ., e_+, e. : \mathcal{T} \ni B \times (B \times B \to B) \times (B \times B \to B) \times B \times B$
Boolean algebra with unities $(B, +, ., e_+, E)$.

Several propositions can be proved in quite a straightforward way:

Proposition 2.1 *Inverses are unique* (UI)

Proof
Let $a \in B, a_1\prime \in B, a_2\prime \in B$ such that
has inverse to $(B, +, e, a, a_1\prime) \wedge$ *has inverse to* $(B, ., e_+, a, a_1\prime) \wedge$
has inverse to $(B, +, e, a, a_2\prime) \wedge$ *has inverse to* $(B, ., e_+, a, a_2\prime)$.
Then:

$$
\begin{aligned}
& a_1\prime \\
= \{E.\}\ & a_1\prime.e \\
= \{I+\}\ & a_1\prime.(a + a_2\prime) \\
= \{D.\}\ & a_1\prime.a + a_1\prime.a_2\prime \\
= \{I.\}\ & e_+ + a_1\prime.a_2\prime \\
= \{E+\}\ & a_1\prime.a_2\prime
\end{aligned}
$$

By symmetry we can conclude $a_1\prime = a_2\prime$
Likewise one can prove the unicity of the unities e_+, e
■

Thus we can define:

def $= : B \to B$
with $\bar{a} = \iota(a\prime : B \,.\, \textit{has inverse to}\ (B, +, e_., a, a\prime) \wedge \textit{has inverse to}\ (B, ., e_+, a, a\prime))$

Corollary 2.2 $\forall(a : B \,.\, \bar{\bar{a}} = a)$ $\qquad (II)$

The following properties (and their proofs) are well known:

$\forall(a : B \,.\, a + a = a)$	$\forall(a : B \,.\, a.a = a)$	(Idp+, Idp.)
$\forall(a : B \,.\, a + e_. = e_.)$	$\forall(a : B \,.\, a.e_+ = e_+)$	(U+, U.)
$\forall(a, b : B \times B \,.\, a + a.b = a)$	$\forall(a, b : B \times B \,.\, a.(a + b) = a)$	(Abs+, Abs.)
$\forall(a, b : B \times B \,.\, a + \bar{a}.b = a + b)$	$\forall(a, b : B \times B \,.\, a.(\bar{a} + b) = a.b)$	(Alt+, Alt.)
$\forall(a, b : B \times B \,.\, \overline{a + b} = \bar{a}.\bar{b})$	$\forall(a, b : B \times B \,.\, \overline{a.b} = \bar{a} + \bar{b})$	(DM+, DM.)
$\forall(a, b, c : B^3 \,.\, a + (b + c) = (a + b) + c)$		(A+)
$\forall(a, b, c : B^3 \,.\, a.(b.c) = (a.b).c)$		(A.)

With the exception of the properties (DM+, DM.) all of these can be proven with an equational transformational proof. Consider for instance the proof of (Idp+):

Proof

$$
\begin{array}{ll}
 & a + a \\
= \{E.\} & (a + a).e_. \\
= \{I+\} & (a + a).(a + \bar{a}) \\
= \{D+\} & a + (a.\bar{a}) \\
= \{I.\} & a + e_+ \\
= \{E+\} & a
\end{array}
$$

■

The most difficult (and also the most powerful) properties to prove are the associativity properties. Here we prove (A+):

Proof

$$
\begin{array}{ll}
 & ((a + b) + c).(a + (b + c)) \\
= \{D.\} & ((a + b) + c).a + (((a + b) + c).b + ((a + b) + c).c) \\
= \{C.\} & a.((a + b) + c) + (b.((a + b) + c) + c.((a + b) + c)) \\
= \{D.\} & (a.(a + b) + a.c) + ((b.(a + b) + b.c) + (c.(a + b) + c.c)) \\
= \{C+\} & (a.(a + b) + a.c) + ((b.(b + a) + b.c) + (c.c + c.(a + b))) \\
= \{Abs.\} & (a + a.c) + ((b + b.c) + (c.c + c.(a + b))) \\
= \{Idp.\} & (a + a.c) + ((b + b.c) + (c + c.(a + b))) \\
= \{Abs+\} & a + (b + c)
\end{array}
$$

In an analog way one proves $(a + (b + c)).((a + b) + c) = (a + b) + c$.
■

The two rules of the Morgan (DM+,DM.) can only be proven indirectly, using the

uniqueness of inverses. We prove (DM+):

Proof

$$
\begin{array}{ll}
 & (a+b)+\bar{a}.\bar{b} \\
= \{Idp., A+\} & (a+\bar{a}.\bar{b})+(b+\bar{a}.\bar{b}) \\
= \{Alt.\} & (a+\bar{b})+(b+\bar{a}) \\
= \{A+, C+\} & (a+\bar{a})+(b+\bar{b}) \\
= \{I+\} & e_{.}+e_{.} \\
= \{Idp+\} & e_{.}
\end{array}
$$

Likewise one proves that $(a+b).\bar{a}.\bar{b} = e_{+}$. Consequently $\bar{a}.b$ is an inverse of $a+b$. By proposition (UI) we may conclude that $\overline{a+b} = a.\bar{b}$, thus finishing the proof of (DM+).

■

The proof of (DM.) goes similarly.

end packet

As we can see the basic axioms, properties and proofs can be formulated in a very elegant way using the Funmath notation and the equational transformational proof style. Remark however, that we had to introduce two symbols e_{+} and $e_{.}$ to denote the unities instead of the usual symbols 0 and 1, normally encountered in texts on boolean algebra. If we would have wanted to use 0 and 1 we would have had to rebind these predefined (numerical) constants, which is undesirable because we want to use 0 and 1 as truth values in reasoning *about* boolean algebras. Note also that we had to denote inverses by quotes and prove their uniqueness before we could introduce the usual notation for inverses ($\bar{}$).

2.3 Group theory

As in the previous section we define the group axioms by introducing some auxiliary functions:

def *associative* : $\mathcal{T} \ni G \times (G \times G \to G) \to \mathbb{B}$
with *associative* $(G, *) = \forall(a, b, c : G^3 \,.\, a * (b * c) = (a * b) * c)$

def *commutative* : $\mathcal{T} \ni G \times (G \times G \to G) \to \mathbb{B}$
with *commutative* $(G, *) = \forall(a, b : G^2 \,.\, a * b = b * a)$

def *has left unit* : $\mathcal{T} \ni G \times (G \times G \to G) \times G \to \mathbb{B}$
with *has left unit* $(G, *, e) = \forall(a : G \,.\, e * a = a)$

def *has right unit* : $\mathcal{T} \ni G \times (G \times G \to G) \times G \to \mathbb{B}$
with *has right unit* $(G, *, e) = \forall(a : G \bullet a * e = a)$

If an algebra has a left unit and a right unit, they must equal. Moreover there is exactly one unit in that algebra:

Lemma 2.3 *has left unit* $(G, *, e_1) \wedge$ *has right unit* $(G, *, e_2) \Rightarrow e_1 = e_2$

Proof

	e_1
$= \{$*has right unit* $(G, *, e_2)\}$	$e_1 * e_2$
$= \{$*has left unit* $(G, *, e_1)\}$	e_2

■

We continue:

def *has unit* : $\mathcal{T} \ni G \times (G \times G \to G) \times G \to \mathbb{B}$
with *has unit* $(G, *, e) =$ *has left unit* $(G, *, e) \wedge$ *has right unit* $(G, *, e)$

def *inverse* : $\mathcal{T} \ni G \times (G \times G \to G) \times G \to \mathbb{B}$
with *inverse* $(G, *, e) = \forall(a : G \bullet \exists(a\prime : G \bullet a * a\prime = e \wedge a\prime * a = e))$

def *group with unit* : $\mathcal{T} \ni G \times (G \times G \to G) \times G \to \mathbb{B}$
with *group with unit* $(G, *, e) =$ *associative* $(G, *) \wedge$ *has unit* $(G, *, e) \wedge$ *inverse* $(G, *, e)$

def *Group* : $\mathcal{T} \ni G \times (G \times G \to G) \to \mathbb{B}$
with *Group* $(G, *) = \exists(e : G \bullet$ *group with unit* $(G, *, e))$

def *AbelianGroup* : $\mathcal{T} \ni G \times (G \times G \to G) \to \mathbb{B}$
with *AbelianGroup* $(G, *) =$ *Group* $(G, *) \wedge$ *commutative* $(G, *)$

packet *Basic Group Theory*
$G \in \mathcal{T}, * \in (G \times G \to G), e \in G \mathbb{A}$*group with unit* $(G, *, e)$ **.**

Inverses are unique.

Lemma 2.4 $\forall(a, i_1, i_2 \in G \bullet a * i_1 = e \wedge i_1 * a = e \wedge a * i_2 = e \wedge i_2 * a = e \Rightarrow i_1 = i_2)$

Proof

	i_1
$= \{$*unit*$\}$	$i_1 * e$
$= \{a * i_2 = e\}$	$i_1 * (a * i_2)$
$= \{$*assoc.*$\}$	$(i_1 * a) * i_2$
$= \{i_1 * a = e\}$	$e * i_2$
$= \{$*unit*$\}$	i_2

■

The previous lemma guarantees the wellformedness of the following definitions:

def $\_^{-1} : G \to G$
with $a^{-1} = \iota(a' : G \,.\, a * a' = e \wedge a' * a = e)$

Powers can also be introduced:

def $\_^{\_} : G \times \mathbf{Z} \to G$
with $a^n = (n = 0)?e\dagger(n > 0)?a * a^{n-1}\dagger(a^{-1})^{-n}$

One can easily prove by induction that the following two properties hold:

$\forall(a, m, n : G \times \mathbf{Z} \times \mathbf{Z} \,.\, a^{m+n} = a^m * a^n)$
$\forall(a, m, n : G \times \mathbf{Z} \times \mathbf{Z} \,.\, a^{mn} = (a^m)^n)$

Using this notation the cyclic undergroup generated by an element can be defined:

def $< \_ >: G \to \mathcal{P}G$
with $< a >= \{a^n \mid n : \mathbf{Z}\}$

We define the notion of subgroups:

def $\_$ *is subgroup* $: \mathcal{P}G \to \mathbb{B}$
with H *is subgroup* $=$ *Group* $(H, *))$

A sufficient condition for a subset to be a subgroup is provided by the following proposition:

Proposition 2.5 $\forall(H : \mathcal{P}G \& H \neq \emptyset \,.\, \forall(a, b : H^2 \,.\, a * b^{-1} \in H) \Rightarrow H$ *is subgroup*$)$

Proof
We check the conditions for a group. Associativity is inherited from G. Let $a, b \in H^2$. Then:

a) $e = \{\text{inv.}\}a * a^{-1} \in \{\text{ass.}\}H$

b) $a^{-1} = \{\text{unit}\}e * a^{-1} \in \{a, \text{ass.}\}H$

c) $a * b = \{\text{inv.}\}a * (b^{-1})^{-1} \in \{b, \text{ass.}\}H$

■
end packet

As we can see, defining functions and proving properties pertaining to group theory do not pose any problems in their reformulation in Funmath.

2.4 Topology

2.4.1 Introduction

Topology is often called "rubber geometry". To the basics of topology lies the notion of nearness of points in a geometric space and how these geometric spaces may be mapped unto each other while keeping points that are close to each other in one space, close to each other after being mapped. We have chosen this example on purpose to show that although the *expressive* power of Funmath is fully adequate, the possibility of the *transformational* reasoning style is somewhat "overstretched" in this case. Many proofs that we will encounter are not equational (although still transformational). We consider showing such a weakness as equally illustrative.

2.4.2 The definition of a topology

A *topologic space* is a set of elements, usually called points, together with a collection of subsets called the *topology* on the set, satisfying the following predicate:

def *Topology* : $\mathcal{T} \ni X \times \mathcal{P}(\mathcal{P}X) \to \mathbb{B}$
with *Topology* $(X, \tau) =$
$\emptyset \in \tau \wedge X \in \tau \wedge$
$\forall(A, B : \tau^2 \cdot A \cap B \in \tau) \wedge$
$\forall(f : Fcod\ \tau \cdot \bigcup f \in \tau)$

In classical textbooks the last proposition is often formulated in the following way: given any family $\{U_\iota : \tau\}_{\iota \in I}$ then $\bigcup U_\iota \in \tau$. Clearly the description in Funmath is much more precise and concise.

Some straightforward examples of topologies are the following:

a) $\forall(X \in \mathcal{T} \cdot$ *Topology* $(X,\ \{\emptyset, X\}))$. This topology is called the *indiscrete* topology.

b) $\forall(X \in \mathcal{T} \cdot$ *Topology* $(X,\ \mathcal{P}X))$. This topology is called the *discrete* topology.

c) If X is a set, (Y, σ) is a topological space and $f \in X \to Y$ then

$$Topology\ (X, \{W : \sigma \cdot \{x : \mathcal{D}f \mid f\ x \in W\}\})$$

This topology is called the topology *induced* by f on X. Of particular interest is the case when $X \subseteq Y$ and f is the identity mapping. Any subset of Y, that is given the induced topology, is called a *topological subspace* of (Y, σ).

d) If (X, τ) is a topological space, Y a set, and $f \in X \to Y$ then

$$Topology\ (Y, \{W : \mathcal{P}Y \mid \{x : \mathcal{D}f \mid f\ x \in W\} \in \tau\})$$

This topology is called the topology *conduced* by f on Y. If f is surjective, it is also called the *quotienttopology*.

2.4.3 Bases of a topology

We call a collection of subsets a *base* for a given topology if every element of that topology equals a union of base elements. A sufficient condition for a collection of subsets of some space to be the base of a topology is that the intersection of any two elements of the collection is the union of (other) elements of this collection and that the union of all elements of the collection equals the whole space.

$$\begin{array}{l}\textbf{def}\quad \mathit{TBase} : \mathcal{T} \ni X \times \mathcal{P}(\mathcal{P}X) \to \mathbb{B} \\ \textbf{with}\ \mathit{TBase}\ (X,\beta) = \\ \qquad (\bigcup id_\beta = X) \wedge \forall(B_1, B_2 : \beta^2 \bullet \exists(f \in \mathit{Fcod}\ \beta : \bigcup f = B_1 \cap B_2 \bullet))\end{array}$$

The following theorem shows that the stated condition is indeed sufficient. Hence we may consider topologies induced by a base (In many cases the topology of a topological space is not described at all, but only its base is given).

Theorem 2.6
$\forall(X,\beta : \mathcal{T} \ni X \times \mathcal{P}(\mathcal{P}X) \bullet \mathit{TBase}\ (X,\beta) \Rightarrow \mathit{Topology}\ (X, \{\bigcup f \mid f : \mathit{Fcod}\ \beta\}))$

Proof
Let $X \in \mathcal{T}, \beta \in \mathcal{P}(\mathcal{P}X)$ such that $\mathit{TBase}\ (X,\beta)$. We check the three conditions for a topology:

1) Consider the empty function ε which satisfies $\varepsilon \in \emptyset \to \beta$. Hence $\varepsilon \in \mathit{Fcod}\ \beta$. Then $\emptyset = \bigcup\varepsilon$. Furthermore $id_\beta \in \mathit{Fcod}\ \beta$ and $X = \bigcup id_\beta$.

2) Given $f, g : (\mathit{Fcod}\ \beta)^2$, define k by:

$$\begin{array}{l}\textbf{def}\quad k : \mathcal{D}f \times \mathcal{D}g \to \beta \\ \textbf{with}\ k\ (u,v) = f\ u \cap g\ v\end{array}$$

Remark:

$$\begin{array}{ll}\models \{\text{triv.}\} & \forall(u,v : \mathcal{D}f \times \mathcal{D}g \bullet f\ u \in \beta \wedge g\ v \in \beta) \\ \Rightarrow \{\text{def } \mathit{Tbase}\} & \forall(u,v : \mathcal{D}f \times \mathcal{D}g \bullet \exists(l : \mathit{Fcod}\ \beta \bullet \bigcup l = f\ u \cap g\ v)) \\ \Leftrightarrow \{\text{def } k\} & \forall(u,v : \mathcal{D}k \bullet \exists(l : \mathit{Fcod}\ \beta \bullet \bigcup l = k\ (u,v))) \\ \Rightarrow \{\text{Choice ax.}\} & \exists(L : \mathcal{D}k \to (\mathit{Fcod}\ \beta) \bullet \forall(u,v : \mathcal{D}k \bullet k\ (u,v) = \bigcup(L\ (u,v))))\end{array}$$

Then:

$$\begin{array}{ll} & \bigcup f \cap \bigcup g \\ = \{\eta - \text{conv.}\} & (\bigcup(u : \mathcal{D}f \bullet f\ u)) \cap (\bigcup(v : \mathcal{D}g \bullet g\ v)) \\ = \{\text{prop. } \cap, \cup\} & \bigcup(u,v : \mathcal{D}f \times \mathcal{D}g \bullet f\ u \cap g\ v) \\ = \{\text{def. } k\} & \bigcup(u,v : \mathcal{D}f \times \mathcal{D}g \bullet k\ (u,v)) \\ = \{\eta - \text{conv.}\} & \bigcup(u,v : \mathcal{D}f \times \mathcal{D}g \bullet \bigcup(L\ (u,v)))\end{array}$$

Since $L\ (u, v) \in Fcod\ \beta$, this proves the condition.

3) Of course, the union of unions of base elements is also a union of base elements.

■

The second condition on a topological base may be reformulated in a more useful way:

Proposition 2.7

$$\forall(X, \beta : \mathcal{T} \ni X \times \mathcal{P}(\mathcal{P}X)\,.\, TBase\ (X, \beta) \Leftrightarrow$$
$$(\bigcup id_\beta = X \wedge \forall(B_1, B_2 : \beta^2\,.\,\forall(q : B_1 \cap B_2\,.\,\exists(B_3 : \beta\,.\,q \in B_3 \wedge B_3 \subseteq B_1 \cap B_2))))$$

Proof
It is sufficient to prove that:

$$\forall(B_1, B_2 : \beta^2\,.\,\forall(q : B_1 \cap B_2\,.\,(\exists(B_3 : \beta\,.\,q \in B_3 \wedge B_3 \subseteq B_1 \cap B_2)) \Leftrightarrow$$
$$\exists(f : Fcod\ \beta\,.\,\bigcup f = B_1 \cap B_2))$$

Let $B_1, B_2 \in \beta^2$

$$\begin{array}{lll}
\Rightarrow: & & \forall(q : B_1 \cap B_2\,.\,\exists(B_3 : \beta\,.\,q \in B_3 \wedge B_3 \subseteq B_1 \cap B_2)) \\
& \Rightarrow \{\text{Choice ax.}\} & \exists(f : B_1 \cap B_2 \to \beta\,.\,\forall(q : B_1 \cap B_2\,.\,q \in f\ q \wedge f\ q \subseteq B_1 \cap B_2)) \\
& \Rightarrow \{\text{logic}\} & \exists(f : Fcod\ \beta\,.\,\bigcup f = B_1 \cap B_2)
\end{array}$$

$$\begin{array}{lll}
\Leftarrow: & & \exists(f : Fcod\ \beta\,.\,\bigcup f = B_1 \cap B_2) \\
& \Rightarrow \{\text{set eq.}\} & \exists(f : Fcod\ \beta\,.\,\bigcup f = B_1 \cap B_2 \wedge \forall(q : B_1 \cap B_2\,.\,q \in \bigcup f)) \\
& \Rightarrow \{\text{def. } \bigcup\} & \exists(f : Fcod\ \beta\,.\,\forall(q : B_1 \cap B_2\,.\,\exists(i : \mathcal{D}f\,.\,q \in f\ i \wedge f\ i \subseteq B_1 \cap B_2))) \\
& \Rightarrow \{\text{logic}\} & \forall(q : B_1 \cap B_2\,.\,\exists(B_3 : \beta\,.\,q \in B_3 \wedge B_3 \subseteq B_1 \cap B_2))
\end{array}$$

■

2.4.4 Topologies derived from a metric

Many topologies are derived from a metric. By introducing a metric on a set one formalizes the notion of a *distance* between points of a set. More formally:

$$\begin{array}{l}
\textbf{def}\ \ \textit{Metric space} : \mathcal{T} \ni X \times (X \times X \to \mathbf{R}) \to \mathbb{B} \\
\textbf{with}\ \textit{Metric Space}\ (X, \rho) = \forall\ (x, y, z : X^3\,. \\
\qquad \rho\ (x, y) \geq 0 \wedge (\rho\ (x, y) = 0 \Leftrightarrow x = y) \wedge \\
\qquad \rho\ (x, y) = \rho\ (y, x) \wedge \\
\qquad \rho\ (x, z) \leq \rho\ (x, y) + \rho(y, z))
\end{array}$$

The last inequality is traditionally known as the *triangle inequality*. A very well known metric space is for instance:

$$\textit{Metric Space}\ (\mathbf{R}^n, x, y : \mathbf{R}^n \times \mathbf{R}^n\,.\,(\sum(i : \square n\,.\,(x\ i - y\ i)^2))^{\frac{1}{2}})$$

packet *Metric spaces and topology*
$X, \rho : \mathcal{T} \ni X \times (X \times X \to \mathbf{R}) \mathbin{\&} \mathit{Metric\ Space}\ (X, \rho)$.

The following definition formalizes the notion of an ϵ neighbourhood:

def $B : X \times \mathbf{R}^+ \to \mathcal{P}X$
with $B\ (q, \epsilon) = \{x : X \mid \rho\ (x, q) < \epsilon\}$

Theorem 2.8 *TBase* $(X, \{B\ (q, \epsilon) \mid q, \epsilon : X \times \mathbf{R}^+\})$

Proof

1) Clearly $X = \bigcup(x : X \,.\, B\ (q, 1))$.

2) Given $B_1 = B\ (q_1, \epsilon_1)$ and $B_2 = B\ (q_2, \epsilon_2)$, we must prove that:

$$\forall(q : B_1 \cap B_2 \,.\, \exists(\epsilon_3 : \mathbf{R}^+ \,.\, B\ (q, \epsilon_3) \subseteq B_1 \cap B_2))$$

Let $q \in B_1 \cap B_2$. Define $\delta = \min\ (\epsilon_1 - \rho\ (q_1, q), \epsilon_2 - \rho\ (q_2, q))$. Clearly $\delta > 0$.
Let $M = B\ (q, \delta)$. Let $z \in M$. Then:

$$\begin{array}{ll} & \rho\ (z, q_1) \\ \leq & \{\text{tr. in.}\} \quad \rho\ (z, q) + \rho\ (q, q_1) \\ < & \{z \in M\} \quad \delta + \rho\ (x_1, q) \\ \leq & \{\text{def. } \delta\} \quad \epsilon_1 - \rho\ (x_1, q) + \rho\ (x_1, q) \\ = & \{\text{triv.}\} \quad \epsilon_1 \end{array}$$

So $z \in M \Rightarrow z \in B_1$. Likewise $z \in M \Rightarrow z \in B_2$.
We conclude that $M \subseteq B_1 \cap B_2$.

Using the last theorem on topology bases we may conclude that the conditions for a topology base are met.
■

The topology induced by a metric is called the *Euclidean* topology.
end packet

2.4.5 Elementary topological notions

Topology deals with subsets and complements of subsets. It is therefore convenient to introduce a shorthand for complements:

packet *Basic topology*
$X, \tau \in \mathcal{T} \times \mathcal{P}(\mathcal{P}X) \mathbin{\&} \mathit{Topology}\ (X, \tau)$.

def $-^c : \mathcal{P}X \to \mathcal{P}X$
with $A^c = X \setminus A$

The elements of the topology τ are called the *open* sets in X. The complements of open sets are called *closed* sets.

def $open : \mathcal{P}X \rightarrow \mathbb{B}$
with $open\ A = A \in \tau$

def $closed : \mathcal{P}X \rightarrow \mathbb{B}$
with $closed\ A = A^c \in \tau$

A subset $U : \mathcal{P}X$ containing a point x is called a *neighbourhood* of x if there exists a $V : \tau$ such that $x \in V \wedge V \subseteq U$.

def $\mathcal{N} : X \rightarrow \mathcal{P}(\mathcal{P}X)$
with $\mathcal{N}\, x = \{U : \mathcal{P}X \mid \exists(V : \tau \,.\, x \in V \wedge V \subseteq U)\}$

By definition any open set containing a point is a neighbourhood of that point. Remark also that if there is a neighbourhood of a point that there is also an open neighbourhood of that point.

The *interior* of a set $A \subseteq X$ is the set of points who have a neighbourhood that is a subset of A.

def $-^\circ : \mathcal{P}X \rightarrow \mathcal{P}X$
with $A^\circ = \{x : X \mid \exists(U : \mathcal{N}x \,.\, U \subseteq A)\}$

Proposition 2.9 $\forall(A : \mathcal{P}X \,.\, A^\circ = \bigcup (U : \tau \mid U \subseteq A))$

Proof

$\subseteq$:

	$x \in A^\circ$
$\Rightarrow$ {def °}	$\exists(U : \mathcal{N}x \,.\, U \subseteq A)$
$\Rightarrow$ {def $\mathcal{N}$}	$\exists(U, V : \mathcal{P}X \times \tau \,.\, x \in V \wedge V \subseteq U \wedge U \subseteq A)$
$\Rightarrow$ {set th.}	$\exists(V : \tau \,.\, x \in V \wedge V \subseteq A)$
$\Rightarrow$ {def $\bigcup$}	$x \in \bigcup (U : \tau \mid U \subseteq A)$

$\supseteq$:

	$x \in \bigcup (U : \tau \mid U \subseteq A)$
$\Rightarrow$ {def $\bigcup$}	$\exists(U : \tau \,.\, x \in U \wedge U \subseteq A)$
$\Rightarrow$ {$U \in \mathcal{N}x$}	$x \in A^\circ$

■

Corollary 2.10

$$\forall(A : \mathcal{P}X \,.\, A^\circ \in \tau \wedge A^\circ \subseteq A \wedge (open\ A \Rightarrow A^\circ = A) \wedge A^{\circ\circ} = A^\circ)$$

Proof
From the proposition follows that $A^\circ \in \tau$. That $A^\circ \subseteq A$ is trivial. Remark that $open\ A \Rightarrow A \subseteq A^\circ$. That $A^{\circ\circ} = A^\circ$ is again trivial.
■

Corollary 2.11

$$\forall(A, B : (\mathcal{P}X)^2 \mathbb{A} A \subseteq B \,.\, \textit{open}\ A \Rightarrow A \subseteq B^\circ)$$
$$\forall(A, B : (\mathcal{P}X)^2 \,.\, A \subseteq B \Rightarrow A^\circ \subseteq B^\circ)$$

The *closure* of a set A is the set of points, whose neighbourhoods all have a nonempty intersection with A.

def $\_^- : \mathcal{P}X \to \mathcal{P}X$
with $A^- = \{x : X \mid \forall(U : \mathcal{N}x \,.\, U \cap A \neq \emptyset)\}$

Proposition 2.12 $\forall(A : \mathcal{P}X \,.\, A^- = \bigcap(G : \mathcal{P}X \mid A \subseteq G \wedge \textit{closed}\ G))$

Proof

$\subseteq$: Let $G \in \mathcal{P}X$ such that $\textit{closed}\ G \wedge A \subseteq G$.
Then:

	$x \in G^c$
$\Rightarrow$ {$G^c \in \tau$}	$G^c \in \mathcal{N}x$
$\Rightarrow$ {$A \subseteq G$}	$G^c \in \mathcal{N}x \wedge G^c \cap A = \emptyset$
$\Rightarrow$ {def. $^-$}	$x \in A^{-c}$

So $x \in A^- \Rightarrow x \in G$.

$\supseteq$: Let $x \in \bigcap(G : \mathcal{P}X \mid A \subseteq G \wedge \textit{closed}\ G)$
Remark that for any $V : \tau$

	$A \cap V = \emptyset$
$\Leftrightarrow$ {setth.}	$A \subseteq V^c$
$\Rightarrow$ {$V \in \tau$}	$A \subseteq V^c \wedge \textit{closed}\ V^c$
$\Rightarrow$ {logic}	$x \in V^e$

Consequently $\forall(V : \tau \,.\, x \in V \Rightarrow A \cap V \neq \emptyset)$.
Also:

	$U \in \mathcal{N}x$
$\Rightarrow$ {def $\mathcal{N}$}	$\exists(V : \tau \,.\, x \in V \wedge V \subseteq U)$
$\Rightarrow$ {remark}	$\exists(V : \tau \,.\, x \in V \wedge V \subseteq U \wedge A \cap V \neq \emptyset)$
$\Rightarrow$ {set th.}	$A \cap U \neq \emptyset$

So $x \in A^-$.

■

Corollary 2.13

$$\forall(A : \mathcal{P}X \,.\, closed\ (A^-) \wedge A \subseteq A^- \wedge (closed\ A \Rightarrow A^- = A) \wedge A^{--} = A^-)$$

Corollary 2.14

$$\forall(A, B : (\mathcal{P}X)^2 \& A \subseteq B \,.\, closed\ B \Rightarrow A^- \subseteq B)$$
$$\forall(A, B : (\mathcal{P}X)^2 \,.\, A \subseteq B \Rightarrow A^- \subseteq B^-)$$

The *boundary* of a set A is the set of points whose neighbourhoods have a nonempty intersection with A and have a nonempty intersection with the complement of A.

def $\partial : \mathcal{P}X \to \mathcal{P}X$
with $\partial A = \{x : X \mid \forall(U : \mathcal{N}x \,.\, U \cap A \neq \emptyset \wedge U \cap A^c \neq \emptyset)\}$

Corollary 2.15 $\forall(A : \mathcal{P}X \,.\, \partial A = A^- \cap A^{c-})$

Proposition 2.16 $\forall(A, B : (\mathcal{P}X)^2 \,.\, (A \cap B)^\circ = A^\circ \cap B^\circ)$

Proof

$\subseteq$:

$$\begin{array}{lll} \models \{corr\ 2.10\} & (A \cap B)^\circ \subseteq A \cap B \\ \Rightarrow \{\text{set th.}\} & (A \cap B)^\circ \subseteq A \wedge (A \cap B)^\circ \subseteq B \\ \Rightarrow \{corr\ 2.11\} & (A \cap B)^\circ \subseteq A^\circ \wedge (A \cap B)^\circ \subseteq B^\circ \\ \Rightarrow \{\text{set th.}\} & (A \cap B)^\circ \subseteq A^\circ \cap B^\circ \end{array}$$

$\supseteq$:

$$\begin{array}{lll} \models \{\text{set th.}\} & A^\circ \cap B^\circ \subseteq A^\circ \wedge A^\circ \cap B^\circ \subseteq B^\circ \\ \Rightarrow \{corr\ 2.10\} & A^\circ \cap B^\circ \subseteq A \wedge A^\circ \cap B^\circ \subseteq B \\ \Rightarrow \{\text{set th.}\} & A^\circ \cap B^\circ \subseteq A \cap B \\ \Rightarrow \{corr\ 2.11\} & A^\circ \cap B^\circ \subseteq (A \cap B)^\circ \end{array}$$

■

The following (threefold) proposition may be proven in the same way:

Proposition 2.17

$$\forall(A, B : (\mathcal{P}X)^2 \,.\, (A \cup B)^- = A^- \cup B^-)$$
$$\forall(A, B : (\mathcal{P}X)^2 \,.\, A^\circ \cup B^\circ \subseteq (A \cup B)^\circ)$$
$$\forall(A, B : (\mathcal{P}X)^2 \,.\, (A \cap B)^- \subseteq A^- \cap B^-)$$

Proposition 2.18 $\forall(A : \mathcal{P}X \,.\, A^{\circ c} = A^{c-})$

Proof

$$\begin{array}{ll} & x \in A^{\circ c} \\ \Leftrightarrow \{\text{def } {}^{\circ,c}\} & \neg\exists(U : \mathcal{N}x \,.\, U \subseteq A) \\ \Leftrightarrow \{\text{logic}\} & \forall(U : \mathcal{N}x \,.\, U \not\subseteq A) \\ \Leftrightarrow \{\text{set th.}\} & \forall(U : \mathcal{N}x \,.\, U \cap A^c \neq \emptyset) \\ \Leftrightarrow \{\text{def } {}^-\} & x \in A^{c-} \end{array}$$

■

end packet

2.4.6 Continuous mappings

We start defining the notion of continuity by introducing two auxiliary definitions:

packet *continuity*
$(X,\tau),(Y,\sigma) : (\mathcal{T} \ni Z \times \mathcal{P}(\mathcal{P}Z))^2 \mathbb{A}$ *Topology* $(X,\tau) \wedge$ *Topology* (Y,σ).
refer *Basic topology* (X,τ)
refer *Basic topology* (Y,σ)

def $-^{\rightarrow} : (X \to Y) \to \mathcal{P}X \to \mathcal{P}Y$
with $f^{\rightarrow}\, U = \{f\, x \mid x : U\}$

def $-^{\leftarrow} : (X \to Y) \to \mathcal{P}Y \to \mathcal{P}X$
with $f^{\leftarrow}\, U = \{x : X \mid f\, x \in U\}$

Two very useful propositions are the following:

Proposition 2.19

$\forall(f,U : (X \to Y) \times \mathcal{P}X \,.\, U \subseteq f^{\leftarrow}\,(f^{\rightarrow}\, U))$
$\forall(f,W : (X \to Y) \times \mathcal{P}Y \,.\, f^{\rightarrow}\,(f^{\leftarrow}\, W) \subseteq W)$

Proposition 2.20

$\forall(f,(A,B) : (X \to Y) \times (\mathcal{P}X)^2 \,.\, A \subseteq B \Rightarrow f^{\rightarrow}\, A \subseteq f^{\rightarrow}\, B)$
$\forall(f,(A,B) : (X \to Y) \times (\mathcal{P}Y)^2 \,.\, A \subseteq B \Rightarrow f^{\leftarrow}\, A \subseteq f^{\leftarrow}\, B)$

whose proof is left to the reader.

def $-$ *continuous at* $-$: $(X \to Y) \times X \to \mathbb{B}$
with f *continuous at* $x = \forall(W : \mathcal{N}(f\, x) \,.\, f^{\leftarrow}\, W \in \mathcal{N}x)$

def $-$ *continuous* : $(X \to Y) \to \mathbb{B}$
with f *continuous* $= \forall(x : X \,.\, f$ *continuous at* $x)$

Theorem 2.21

$\forall(f : X \to Y \,.$
$\quad f$ *continuous* $\Leftrightarrow$
$\quad \forall(W : \mathcal{P}Y \,.\, open\ W \Rightarrow open\ (f^{\leftarrow}\, W)) \Leftrightarrow$
$\quad \forall(W : \mathcal{P}Y \,.\, closed\ W \Rightarrow closed\ (f^{\leftarrow}\, W)) \Leftrightarrow$
$\quad \forall(A : \mathcal{P}X \,.\, f^{\rightarrow}\,(A^{-}) \subseteq (f^{\rightarrow}\, A)^{-})\ )$

Proof
We number the parts of this theorem as 1 through 4 and prove the equivalences in the following way:

$1 \Rightarrow 2)$

$$\begin{array}{ll} & W \in \sigma \\ \Leftrightarrow \{\text{def } \mathcal{N}\} & \forall(y : W \,.\, W \in \mathcal{N}y) \\ \Rightarrow \{f \textit{ continuous}\} & \forall(x : f^{\leftarrow} W \,.\, f^{\leftarrow} W \in \mathcal{N}x) \\ \Rightarrow \{\text{trivial}\} & f^{\leftarrow} W \in \tau \end{array}$$

$2 \Leftrightarrow 3)$ Trivial.

$3 \Rightarrow 4)$ Let $A \in \mathcal{P}X$.

$$\begin{array}{ll} \models \{\text{prop 2.19}\} & A \subseteq f^{\leftarrow}(f^{\rightarrow} A) \\ \Rightarrow \{\text{corr 2.13}\} & A \subseteq f^{\leftarrow}((f^{\rightarrow} A)^{-}) \\ \Rightarrow \{3, \text{corr 2.14}\} & A^{-} \subseteq f^{\leftarrow}((f^{\rightarrow} A)^{-}) \\ \Rightarrow \{\text{prop 2.20}\} & f^{\rightarrow}(A^{-}) \subseteq f^{\rightarrow}(f^{\leftarrow}((f^{\rightarrow} A)^{-})) \\ \Rightarrow \{\text{prop 2.19}\} & f^{\rightarrow}(A^{-}) \subseteq (f^{\rightarrow} A)^{-} \end{array}$$

$4 \Rightarrow 1)$ This part of the proof is the actual difficult one.
Let $x \in X$. Let $W \in \mathcal{N}(f\,x)$.
There exist a $W\prime \in \sigma$ with $f\,x \in W\prime \wedge W\prime \subseteq W$.
Let $A = (f^{\leftarrow} W\prime)^{c}$. Then:

$$\begin{array}{ll} & A^{-c} \\ \subseteq \{\text{corr 2.13}\} & A^{c} \\ = \{\text{def } A\} & f^{\leftarrow} W\prime \\ \subseteq \{\text{prop 2.20}\} & f^{\leftarrow} W \end{array}$$

Also:

$$\begin{array}{ll} \models \{\text{def } A\} & f^{\rightarrow} A \subseteq W\prime^{c} \\ \Rightarrow \{\textit{closed } (W\prime^{c})\} & (f^{\rightarrow} A)^{-} \subseteq W\prime^{c} \\ \Rightarrow \{4\} & f^{\rightarrow}(A^{-}) \subseteq W\prime^{c} \\ \Rightarrow \{f\,x \in W\prime\} & x \in A^{-c} \end{array}$$

Apparently $x \in A^{-c} \wedge A^{-c} \in \tau \wedge A^{-c} \subseteq f^{\leftarrow} W$.
We conclude that $f^{\leftarrow} W \in \mathcal{N}x$.

■

We call a function *open* if it maps open subsets unto open subsets. Likewise a function is called *closed* if it maps closed subsets unto closed subsets:

def $-$ *is open* $: (X \to Y) \to \mathbb{B}$
with f *is open* $= \forall(U \in \mathcal{P}X \,.\, \textit{open } U \Rightarrow \textit{open } (f^{\rightarrow} U))$

def $-$ *is closed* $: (X \to Y) \to \mathbb{B}$
with f *is closed* $= \forall(U \in \mathcal{P}X \,.\, \textit{closed } U \Rightarrow \textit{closed } (f^{\rightarrow} U))$

We call a function f a *homeomorphism* if it is bijective and if f and f^{-1} are both continuous. Equivalent is the following definition:

def — *is a homeomorphism* : $(X \to Y) \to \mathbb{B}$
with f *is homeomorphism* = f *is bijective* $\wedge$ f *continuous* $\wedge$ f *is open*

end packet

A homeomorphism is therefore a topology preserving bijective mapping between two topological spaces. A homeomorphism will map open subsets unto open subsets, closed subsets unto closed subsets and boundaries unto boundaries. Two topological spaces are called *homeomorphic*, if there exists a homeomorphism between the two spaces.

Two immediate consequences of the previous theorem are the two following corollaries:

Corollary 2.22

If $(X_1, \tau_1), (X_2, \tau_2)$ *and* (X_3, τ_3) *are topological spaces and* $f_1 \in X_1 \to X_2$ *and* $f_2 \in X_2 \to X_3$ *are continuous, then* $f_2 \circ f_1 \in X_1 \to X_3$ *is also continuous.*

Corollary 2.23

If (X, τ) *is a topological space,* $f \in X \to Y$, f *is surjective and* Y *is given the quotient topology, then* f *is continuous.*

2.4.7 Conclusions

As we have seen, describing topology in Funmath goes well, especially those parts in which arbitrary collections of subsets occur gain in clarity and conciseness by using suitable higher order functions. Proofs in this section were always transformational but almost never equational. This is probably due to the fact that the axioms of topology are not equational but implicational. This is in contrast with axioms describing algebras. However, the author finds proofs written in the transformational style much easier to read than those in ordinary textbooks on topology, since they indicate much clearer what property is used in every proofstep.

Chapter 3

Informal Introduction to the language Glass

Children yet, the tale to hear
Eager eye and willing ear
Lovingly shall nestle near

In a Wonderland they lie
Dreaming as the days go by
Dreaming as the summers die

Ever drifting down the stream
Lingering in the golden gleam
Life, what is it but a dream.
Through the Looking-glass *Lewis Carroll*

3.1 Introduction

Glass (General language for system semantics) is a language for describing lumped systems (physically existing systems, composed of discrete objects having a discrete interface), although its present usage is limited to electronic systems. The ideas for this language and its "describing environment" (as opposed to the term "programming environment") are based upon systems semantics [Bou88].

Inspired by the Algol-68 tradition [Wij76], [Lin80], the language is documented in two reports, namely the "Language reference manual" [Seu91] and the "Informal introduction to the language Glass", which was published first in [Seu90]. The first of these describes the precise syntax, typing system, and macro expansion rules of the language. The latter is intended to introduce the language Glass in an informal way to those who have not seen any Glass description before. This chapter is a revision of that original report. Thus we will pay more attention to describing the ideas behind the language and elucidating the language concepts by many examples than to describing

its exact syntax. The reader must also be aware that this chapter does not intend to cover every construct in the language.

Glass also has two language representations, namely the *reference language* and the *implementation language*. The first is also intended as *publication language* and is therefore used throughout this thesis in all of the examples. However the lexical symbols are not always available in standard Ascii character sets, and hence the publication reference language is not suited for automated interpretation on the basis of Ascii strings. For this purpose the implementation language may be used. In appendix B of this thesis the implementation language equivalents of the examples presented in this chapter are given.

3.2 Systems semantics

"When I use a word,", Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean, neither more nor less"
Through the Looking-glass *Lewis Carroll*

3.2.1 Motivation

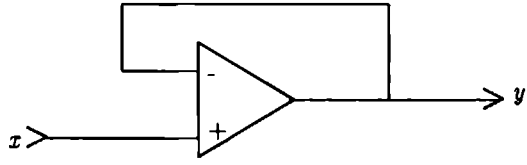
When designing hardware the designer has to consider various aspects of hardware such as structure, behaviour, cost, layout, etc. In general he will try and obtain a design, whose behaviour satisfies the design specification given beforehand, while minimizing the cost of the design and the amount of area needed for the layout of the circuit.

Technical systems such as electric circuits are essentially non algorithmic: the electrical phenomena in circuits are not computational processes but physical processes, whose behaviour is governed by system equations based (essentially) on Maxwell's equations for the electromagnetic field. A similar situation holds for the trajectory of the moon in the joint gravitational field of the sun and the earth. It is absurd to say that the moon executes Newton's algorithm, although the general laws of gravity and kinematics control its motion [New86]. As a paradigm for description and modelling, algorithms express at most a restricted simulation oriented view of a discrete time model of behaviour.

Many of the current hardware description languages are quite difficient. Oft only one aspect of the design is described, whereas other aspects must be described in another hardware description language or in a natural language. Or, in one language several descriptions must be given for different aspects of the design. VHDL [IEEE88] is an example of this class. When describing hardware in VHDL, one describes the structure of the design as well as a "description" of its behaviour by giving an algorithm for an event driven simulation. Again not every aspect of the design is covered by the description.

Moreover one may ask whether discrete event simulation is a correct way to de-

scribe hardware behaviour. The following example shows that simulation may be quite unrealistic. Consider a voltage follower built with an operational amplifier:



We may describe the behaviour of this opamp by the following system equation:

$$y = A.(x - y)$$

In practical circuits the amplification factor A is of the order of 10^5. Now suppose that we want to simulate the behaviour of this circuit using the following formula:

$$y_{n+1} = A.(x_n - y_n)$$

where we use as initial values $x_0 = y_0 = 0$. If we then apply a step function to the input x we may observe divergent behaviour of the output y, which certainly does not comply to the actual behaviour of the system. However if we solve the sytem equation symbolically we obtain:

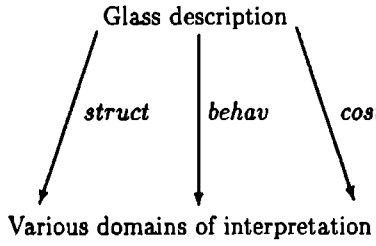
$$y = \frac{A}{1+A}.x$$

which is a far better description of the behaviour of the voltage follower.

3.2.2 The principle of systems semantics

The basic principle of systems semantics is that of notational economy: one describes hardware formally using appropriate syntax and scoping rules based upon the Euler notation of functions. Only *one* language is used to describe formally the decomposition of a system into subsystems and their connectivity. The language Glass is based upon this principle.

Such a formal description may then be assigned a meaning (interpretation) by a *semantic function*. A semantic function is a function from the set of legal Glass descriptions to a certain domain of interpretation. Descriptions in Glass therefore have no *a priori interpretation* at all: they are mere pieces of texts obeying the Glass syntax rules. An interpretation of a Glass description can only be derived by applying a semantic function to it. By applying several of these semantic functions to one description one may obtain various aspects of the described circuit such as structure, cost, behaviour etc. In this way one can also study various behavioural models of the described circuit without having to change the actual description. This amounts to a *multiview* of the system described:



In this way, Glass is a single system description language, with multiple interpretations.

A semantic function should, if possible, be *compositional*, that is, it expresses the meaning of a composite system in terms of the meanings of the constituents. The meaning of elementary or primitive constituents, called *atoms* in Glass, must therefore be coded into the semantic function according to the actual model of interpretation implemented by the semantic function. The writer of the semantic function is therefore responsible for fixing the properties of the atoms according to the implemented model.

The primary aim of Esprit project 881, "Forfun" (started May 1986, ended May 1990) was a feasability study of the concept of systems semantics and (if possible) the development of an experimental prototype "describing environment". In this project the language Glass [Seu90] has been developed and implemented. Several semantic functions for the digital and analog domain have also been implemented together with a number of support tools.

3.3 Basic language design

3.3.1 Terminology

Descriptions in Glass, in their structural interpretation, define systems in terms of subsystems and their connectivity. Not all interconnections are of the same nature: systems may be coupled electrically (by wires), magnetically, pneumatically, optically (by glass fibers), etc. We will therefore use the term *connection* in a very general sense, not only for electric connection, but also for connection via magnetics, optics etc. We will also use the word *terminal* in a general sense, being an interface between (sub)systems.

Two systems may also be multiply connected: the interface between the two may be decomposed into sets of subconnections. A system description is therefore characterized not only by its decomposition, but also by its external interface. The specification of such an external interface is called the *type* of a system.

Basic interconnections are introduced by means of so called *base type* declarations, which serve to introduce a name for an elementary connection. In all of our examples there will be only one basetype E, which indicates the type of a wire.

3.3.2 Directionality of systems

Systems may be differentiated into three large classes namely *directional*, *adirectional* and *hybrid* ones. In directional systems there is a clear signal flow or flow of information from a certain set of terminals (called inputs) to another set of terminals (outputs). Digital circuits from the gate level and upwards typically belong to this class. In the adirectional systems such a clear flow of information is missing. Analogue circuits at the component level (R, L, C, active devices) typically belong to this class. Hybrid systems (systems that are neither purely directional nor purely adirectional but more a kind of mixture of both) also exist, as we will see later.

This differentiation can be made for the interface of a system as well as for its decomposition. A system may well be directional for the outside world whereas it can be composed of adirectional subsystems. For both parts of a description special syntactic constructs may be used to indicate directionality. Since the directional subset of Glass is closer to usual mathematical constructs than its adirectional subset, we will discuss the description of directional systems (digital circuits) first.

3.3.3 Kernel and macro language

One of the design concerns of the language is the ability to support the description of regular structures, as they are often encountered in VLSI design.

The language therefore contains a macro expansion mechanism, with which we can describe regular systems. It is also useful for obtaining compact descriptions. A macro describes only how the descriptions that make up a regular circuit must be generated. It does not have a direct meaning nor can a semantic function directly ascribe a meaning to it. Meaning can, in principle, only be attributed to the expanded form.

Glass therefore contains a clearly distinguishable subset called the *kernel language*, in which only descriptions of proper circuits (descriptions that actually have a structural interpretation) may be described. Semantic functions are only applied to descriptions in the kernel language. Glass descriptions are expanded by the macro expander in the describing environment, thus removing all macro constructs, before semantic functions may be applied to them.

To steer the actual expansion the macro language is in fact a full fledged functional language, with which one may calculate, evaluate conditionals, parametrize systems, etc. Recursion may be used to replicate subsystems. These constructs are also removed by the macro expansion mechanism.

Moreover the macro language supports special syntactic constructs to parametrize descriptions. We may consider three kinds of parametrization:

- Parametrization of systems
  It is evident that macro expansion must result in finite descriptions (Infinite circuits have no structural interpretation). Macros may have extra parameters that can be used to indicate the size of the resulting description. Moreover

systems may be used as parameters in higher order descriptions (in analogy of higher order functions in functional programming languages).

- Parametrization of components
  It is often useful to give values to a discrete component (say a resistor) in a design. Since the thus given parameters actually provide extra information of the component to semantic functions, these parameters may occur in kernel Glass descriptions, but only as constant parameters of atomic components.

- Parametrization to support abstraction
  This parametrization is typically used to avoid describing special initialization circuitry, which can also be described in later stages of the design. One may think for instance of the initial state of flipflops, etc. Again, after macro expansion these parameters must have been evaluated into constant parameters of atomic components.

3.4 Directional systems

3.4.1 A very simple system

"So it does!" said Pooh. "It goes in!"
"So it does!" said Piglet. "And it comes out!"
"Doesn't it?" said Eeyore.
"It goes in and out like everything."
Winnie-the-Pooh A.A. Milne

Consider the following description:

basetype E;

atom *not* $\in E \Rightarrow E$;

The first sentence of this Glass text introduces E as a basetype, which will stand for the type of a wire, as explained earlier. The second introduces an atomic component called *not*. Atomic system definitions can not be decomposed into subsystems as they stand for primitive components, but their external interface *must* be specified. We will call such a specification of the interface a *declaration*. In the declaration we see a first example of a type constructor namely the $\Rightarrow$. In general $U \Rightarrow V$ stands for the set of directional systems having terminals of type U as input and terminals of type V as output. Thus *not* is declared to be an element of the set of directional systems having one input of type E and one output of type E.

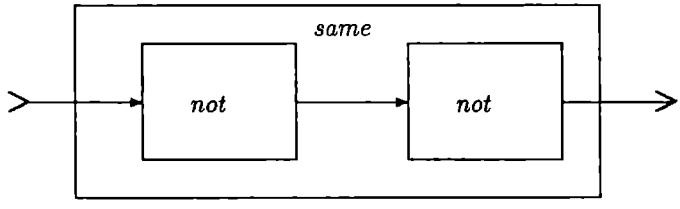
The following example describes a very simple system:

def *same* $\in E \Rightarrow E$;
same x = *not* (*not* x);

It is the first example of a composite system definition thus far. A composite system definition consists of a declaration (the type declared must of course be valid for the description), and the description of its decomposition into subsystems. In directional systems this decomposition looks syntactically like a function definition in mathematics and functional programming languages, where $f\ u$ stands for the application of the function f to the parameter u. This notation is also used in Glass.

At the left hand side of the equals sign one can see the system name and a name of its input parameter. The system name is visible outside the description and can be used in other descriptions. The parameter name is bound and is a purely local name (internal to the definition) for the input connection.

At the right hand side stands an expression. In this expression application of a system to a subexpression means the connection of output to input or if that subexpression is a name, connection of that terminal to a input. In this way *same* is described as a system with one input terminal named x, and one output terminal which is connected to a row of two *not*s fed by x:



As a convention when presenting the structural interpretation of descriptions, we will indicate the directionality of the terminals in the following way:

Mark that the parentheses in the right hand side of the definition are really necessary: without them one would read this expression as the application of a system to a system applied to x (i.e. (*not not*) x), which clearly is faulty Glass. Check the typing for instance: From the expression *not* e, we derive that e must have type E. In the expression *not not* the second *not* has type $E \Rightarrow E$.

As a general rule, parentheses ('(' and ')') are used for indicating priority.

3.4.2 Multiple connections

Many systems have more than only one input. In Glass this is represented by compound connections. The type of a compound connection is introduced by the (infix) $\times$ type constructor. Thus $E \times E$ is the type of a bundle of two wires. Although this notation is very similar to the notation used in mathematics to indicate the cartesian products,

the reader must remain aware of the fact, that Glass descriptions do not have an a priori interpretation. It may well be that a semantic function maps such a type product on a single set instead of a product of sets.

A shorthand may be used to indicate repeated type products: U^k stands for the k-fold type product $U \times U \times \ldots \times U$, provided $k \geq 2$. The following example introduces some well known atomic components with two inputs.

basetype E;

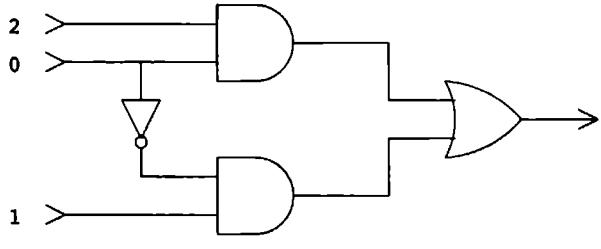
atom $and \in E \times E \Rightarrow E$,
$or \in E \times E \Rightarrow E$,
$xor \in E \times E \Rightarrow E$;

Two terminals of type E, say x and y, can be combined into a terminal of type $E \times E$ by forming a pair $\langle x, y \rangle$. In general if we have a set of terminals $a_1, a_2, \ldots, a_n$ so that a_i is a terminal of type U_i then $\langle a_1, a_2, \ldots, a_n \rangle$ is a terminal of type $U_1 \times U_2 \times \ldots \times U_n$. Let us continue the description:

atom $not \in E \Rightarrow E$; /* *we have met this one before!* */

def $select \in E \times E \times E \Rightarrow E$;
$select\ \langle s, a, b \rangle = or\ \langle and\ \langle not\ s, a \rangle, and\ \langle s, b \rangle \rangle$;

The structural interpretation of this description is represented by the following schematic:



When presenting the structural interpretation of systems with compound inputs we indicate the subinputs by numbers counting from 0 in the order of the subinputs in the Glass description. This is done recursively for compound subinputs, separating the numbers by points.

Tupling (making a tuple) is allowed in the formal arguments of a system definition as well as in expressions. An example of this is the following description:

def $halfadder \in E \times E \Rightarrow E \times E$;
$halfadder\ \langle x, y \rangle = \langle xor \langle x, y \rangle,\ and\ \langle x, y \rangle \rangle$;

3.4.3 Anonymous systems

In all our previous examples the described systems had names. Sometimes, however, it is useful to describe *anonymous systems*. A nameless directional system is introduced by a λ-abstractor. With

$$\lambda\ \langle a, b\rangle.not\ (or\ \langle a, b\rangle)$$

we mean an anonymous system with two inputs, consisting of an *or* and a *not* gate. The type of this anonymous system $(E \times E \Rightarrow E)$ must be inferred from the types of the *or* and the *not* gate.

These λ-abstractors may appear anywhere in expressions. In particular, the right hand side of a definition may consist of a single λ-abstractor. In fact the following two definitions are completely equivalent:

def $ornot \in E \times E \Rightarrow E;$
$ornot\ \langle a, b\rangle = not\ (or\ \langle a, b\rangle);$

def $ornot \in E \times E \Rightarrow E;$
$ornot = \lambda\ \langle a, b\rangle.not\ (or\ \langle a, b\rangle);$

The first definition can be seen as a *syntactically sugared* version of the second definition which is in fact the way in which the Glass parser in the describing environment treats this description.

3.4.4 Feedback and Fanout

But Eeyore wasn't listening.
He was taking the balloon out,
and putting it back again
as happy as could be.
Winnie-the-Pooh *A.A. Milne*

In many systems *feedback* may be encountered. In the digital field feedback is used to build sequential circuits, synchronous and asynchronous ones. In the analog domain, feedback is used, for instance, to reduce the distortion in amplifiers or to control a plant.

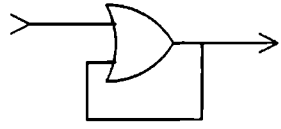
In Glass *feedback* and also *fanout* (that is, the internal connection of several inputs to the same subsystem output) may be described by a **where**-*clause*, which introduces *local* definitions. Multiple local definitions must be separated by semicolons. Consider for instance the following definition:

basetype $E;$

atom $or \in E \times E \Rightarrow E;$

```
def  SetOnce ∈ E ⇒ E;
     SetOnce i = o where o = or ⟨i, o⟩ endwhere;
```

which has as structural interpretation:

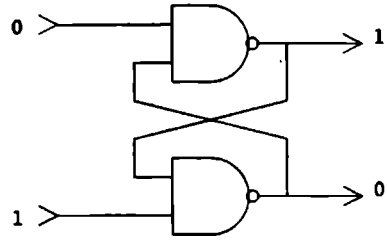


Another illustrative example is the description of the *reset-set-flipflop*:

```
atom  nand ∈ E × E ⇒ E;

def  RSFF ∈ E² ⇒ E²;
     RSFF ⟨R', S'⟩ = ⟨q, q'⟩
         where
           q = nand ⟨S', q'⟩;
           q' = nand ⟨R', q⟩
         endwhere;
```

whose structural interpretation can be represented by this schematic:

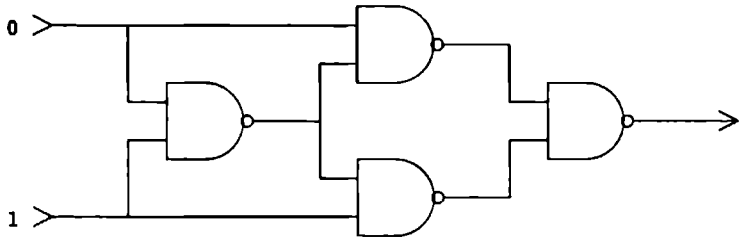


Notice the usage of primes in the description as lexical symbols in identifiers. They do not indicate not gates (Glass is an uninterpreted language), but the writer may use them in identifiers to suggest certain properties to the reader of the description. In this example they suggest that the inputs R' and S' are active low inputs.

An example of expressing fanout with the **where** construct is the following somewhat unusual implementation of an exclusive or:

```
def  my_xor ∈ E × E ⇒ E;
     my_xor ⟨x, y⟩ = nand ⟨nand ⟨b, x⟩, nand ⟨b, y⟩⟩
         where b = nand ⟨x, y⟩; endwhere;
```

whose structural interpretation is represented by this schematic:



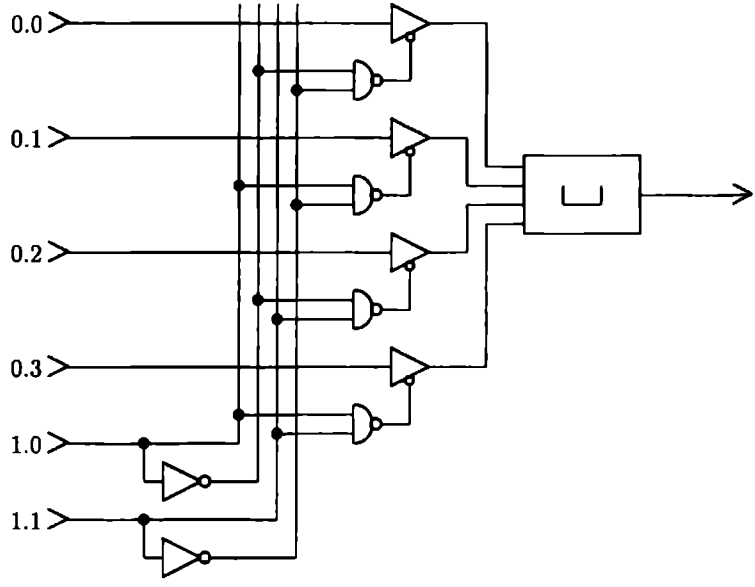
3.4.5 Description of tristate interfaces

The syntax of Glass does not allow the connection of two or more outputs to each other. To describe tristate buffers and buses one must declare them as special directional atoms. Properly defined semantic functions may then yield the correct behavioural properties for these atoms. Consider the following description of four outputs that are gated onto one common bus:

atom $not \in E \Rightarrow E$,
$nand \in E \times E \Rightarrow E$,
$tbuf \in E \times E \Rightarrow E$,
$join_4 \in E^4 \Rightarrow E$;

def $busif \in E^4 \times E^2 \Rightarrow E$
$busif\ \langle\langle s_0, s_1, s_2, s_3\rangle, \langle a_0, a_1\rangle\rangle = join_4\ \langle t_0, t_1, t_2, t_3\rangle$
where
$t_0 = tbuf\ \langle nand\ \langle a_0', a_1'\rangle,\ s_0\rangle$;
$t_1 = tbuf\ \langle nand\ \langle a_0, a_1'\rangle,\ s_1\rangle$;
$t_2 = tbuf\ \langle nand\ \langle a_0', a_1\rangle,\ s_2\rangle$;
$t_3 = tbuf\ \langle nand\ \langle a_0, a_1\rangle,\ s_3\rangle$;
$a_0' = not\ a_0$;
$a_1' = not\ a_1$;
endwhere

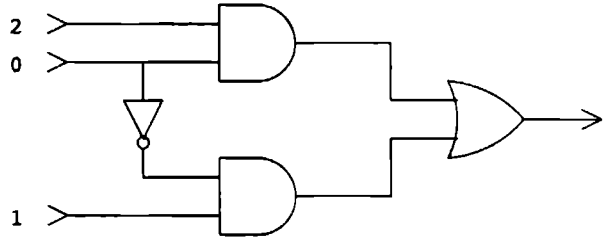
which has as structural interpretation:



3.5 Some simple semantic functions

In this section we will introduce some very simple semantic functions. Remember, however, that neither semantic functions nor their implementation are part of the language. They are only applied to Glass texts. Those interested in the actual implementation of the semantic functions are referred to later chapters of this thesis.

The first semantic function we introduce is *struct*, which yields the structural interpretation of a description. Applying *struct* to the *select* description of the previous section yields its structure, which can be represented by the following schematic:



Another semantic function yields the static behaviour of the described system, i.e. the input/output function for instantanous values (which clearly is meaningful only for

memoryless systems). For digital systems the instantaneous values usually are the values 0 and 1. The corresponding semantic function *simplex* maps a circuit description to a boolean function. For instance, the image of *select* under *simplex* is a boolean function of type $\mathbb{B}^3 \rightarrow \mathbb{B}$ with $\mathbb{B} = \{0, 1\}$, describing the statical input/output mapping realized by that circuit, e.g.

$$(simplex\ [\![select]\!])(s, a, b) = (\neg s \wedge a) \vee (s \wedge b)$$

A third example of a semantic function is the *cost* function, which yields the cost of a description based on a certain cost model. For instance, if the cost model used is the number of gates in the circuit, then $cost\ [\![select]\!] = 4$.

3.6 Adirectional and hybrid systems

3.6.1 Properly adirectional systems

In adirectional systems, by definition, there is no clear direction of signal flow or flow of information. Such a system may have terminals connecting it to the outside world but it is not meaningful to designate input or output. Consider for instance a resistor: it has two terminals to connect it with other components, but current may flow into a terminal, and at the same time out of the other (Kirchhoffs law), but the direction of the current may be entirely unrelated to (the direction of) the signal flow.

As all system definitions must contain a declaration, it must be possible to specify that a circuit is adirectional. With $[U]$ we mean the set of all adirectional systems having terminals of type U. So a resistor has type $[E \times E]$. Let us introduce some typical adirectional systems:

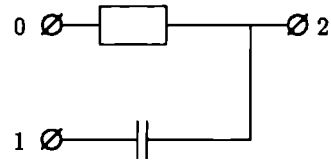
basetype E;

atom $R \in [E \times E]$,
$C \in [E \times E]$,
$NPN \in [E \times E \times E]$; /* *e.g. emittor, base, collector* */

As there is no output in an adirectional system, the right hand side of a composite system definition also has an appropriate form. An adirectional system is described in terms of an unordered collection of all its constituents together with their interconnections. We will call this collection an *appset* (which has type **Appset**). In an appset connectivity is indicated by using common symbols. Because we do not want that all local connections are visible to the outside, names are bound within the appset unless they are already bound outside the appset (for instance as formal parameter of the system). A simple example of an appset is the following description:

def $RCnet \in [E \times E \times E]$;
$RCnet\ \langle a, b, c\rangle = \{\ R\ \langle a, c\rangle,\ C\ \langle b, c\rangle\}$;

with the following structural interpretation



In the presentation of the structural interpretation of adirectional descriptions adirectional terminals are indicated as follows

Appsets may contain system applications, appsets and so-called synonym declarations

A synonym declaration is the application of a special symbol $\star$ to a collection of nets indicating the proper interconnection of these nets These nets must of course be composed in the same way, so all must have the same type An example of an appset with a synonym declaration in it is the following

def *RCnet2* $\in [E \times E \times E]$,
RCnet2 $\langle a, d, x \rangle = \{\ R\ \langle a, x \rangle,\ C\ \langle d, y \rangle,\ \star\ \langle x, y \rangle \}$,

which has the same *structural interpretation* as the previous description

3.6.2 Global connections in descriptions

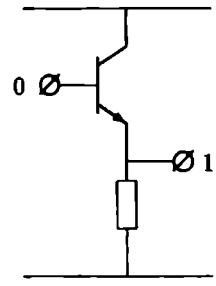
In analog systems components are often connected to global nets such as power supplies Consider for instance a NPN-transistor whose emittor is grounded One way of describing connections to global nets is to declare these nets as atomic components having only one (adirectional) terminal

atom *Supply* $\in [E]$,
Gnd $\in [E]$,

An example of their use is the following description of a one-transistor voltage-follower

def *VoltageFollower* $\in [E \times E]$,
VoltageFollower $\langle in, out \rangle =$
$\{\ NPN\ \langle out, in, plus \rangle,$
$R\ \langle out, gnd \rangle,$
Supply plus,
Gnd gnd$\}$,

which has the following structural interpretation



3.6.3 Anonymous adirectional systems

As in directional systems it is possible to describe *anonymous adirectional* systems. An anonymous adirectional system is introduced by a σ-abstractor. With

$$\sigma\langle a,b\rangle.\{\ R\ \langle a,c\rangle,\ C\ \langle c,b\rangle\}$$

we mean an adirectional system with two terminals (i.e. having type $[E \times E]$) consisting of a resistor and a capacitor in series. Again the type of the construct must be deduced from the types of its constuents.

3.6.4 Partially adirectional systems

In this paragraph, we will encounter circuits that are partially adirectional, that is, some of the terminals of a system are truly adirectional, whereas others can be seen as inputs or as outputs. The directionality of a terminal can be indicated in its type: $?U$ is the type of an input terminal of type U, and $!U$ is the type of an output terminal of type U. Consider, for instance, the declaration of an atomic monostable flipflop, which has one input, two terminals to connect it to an external timing capacitor, and one output:

basetype E;

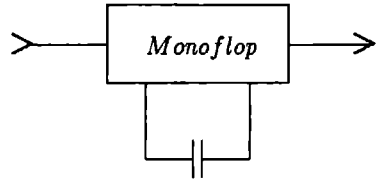
atom *Monoflop* $\in [?E \times E \times E \times !E]$;

Notice that the type $U \Rightarrow V$ is equivalent to the type $[?U \times !V]$. Thus $E^2 \Rightarrow E$ is equivalent to $[?(E^2) \times !E]$. This equivalence may be propagated through products (and their shorthands): $?(U_1 \times U_2 \times \ldots \times U_n)$ is equivalent to $?U_1 \times ?U_2 \times \ldots \times ?U_n$. So $E^3 \Rightarrow E^2$ is equivalent to $[(?E)^3 \times (!E)^2]$.

In the same way, one can decompose directional systems in a set of (partially) adirectional components. Consider for instance the following description of a delay element:

def *delay* $\in E \Rightarrow E$;
delay $\langle in, out\rangle = \{$ *Monoflop* $\langle in, x, y, out\rangle, C\ \langle x, y\rangle\}$;

hich has the following structural interpretation, represented as schematic:



.6.5 Two illustrative hybrid examples

emark that the concepts introduced in the previous paragraph can also be used t escribe systems that are directional to the outside world but internally built wit lirectional components. An example of such a system is an amplifier: it has on iput and one output, but internally it is built with adirectional elements such a ansistors, resistors, capacitors, etc.

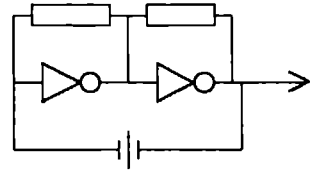
An example of such a system is the following description of a crystal oscillator:

atom $R \in [E \times E]$,
$Xtal \in [E \times E]$;

atom $not \in E \Rightarrow E$

def $Xosc \in [!E]$;
$Xosc\ z =$
$\{\ R\ \langle x, y\rangle,$
$R\ \langle y, z\rangle,$
$Xtal\ \langle x, z\rangle,$
$not\ \langle x, y\rangle,$
$not\ \langle y, z\rangle$
$\}$

hich has the following structural interpretation:



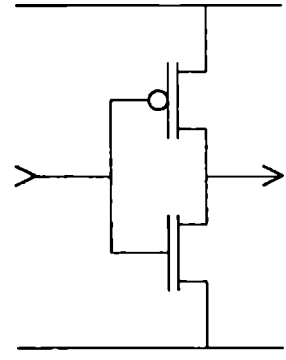
nother example can be seen in the following description of an invertor in CM

```
atom  Nenh ∈ [E × E × E],      /* gate, source, drain */
      Penh ∈ [E × E × E];      /* idem */

atom  Vdd ∈ [E],
      Gnd ∈ [E];

def   CMosInvertor ∈ E ⇒ E;
      CMosInvertor ⟨in, out⟩ =
          { Penh ⟨in, plus, out⟩,
            Nenh ⟨in, gnd, out⟩,
            Vdd plus,
            Gnd gnd};
```

which has the following *structural interpretation*:



3.7 Typing and parametrization

3.7.1 Function types

Since the macro language is a full fledged functional (programming) language, it must be possible to introduce functions in Glass. Introducing a new type constructor, the function arrow $\rightarrow$, we write $U \rightarrow V$ to denote the set of all functions with domain U and codomain V. In the case of macros, it is the type of macros with a formal argument from U, giving a result in V after macro expansion.

The reader is warned that he should not confuse the two arrows $\rightarrow$ and $\Rightarrow$. The first expresses a function type, whereas the second expresses a system type (and has higher precedence). An illustrative example hereof can be seen in the type of the following macro:

```
mac  triple ∈ E ⇒ E → E ⇒ E;
     triple A in = A (A (A in));
```

3.7.2 Other primitive types

As mentioned earlier it is convenient to be able to calculate in Glass. We therefore introduce some new types and constants of these types in Glass :

- **Int** (the set of integer numbers)
- **Float** (the set of floating point numbers)
- **String** (the set of character strings)
- **Bool** (the set of booleans)

The language also contains several well known predefined operators for these sets like $+$, $-$, $\times$, **Div** and **Mod**, etc, which are evaluated by the macro expander.

3.7.3 Parametrized atoms

In subsection 3.6.1 it was suggested to declare a resistor as $R \in [E \times E]$. This is not sufficiently general, since one would like to indicate the values of resistors as well. Moreover, the specification of values may provide extra information for semantic functions. The $\rightarrow$ and primitive types introduced previously provide us a way:

atom $R \in$ **Float** $\rightarrow [E \times E]$

declares that R, applied to some floating point number (e.g. $R\,4.7$), is an adirectional atom of type $[E \times E]$.

Remark that these parametrized atoms are not macros, but are also part of the kernel language, with the restriction that the actual parameters of these atoms must have been evaluated into constants by the macro expander (Otherwise, they do not have a structural interpretation).

3.7.4 Compound types

Compound types in Glass adhere to the same convention as cartesian products in Funmath: they are actually functions. By the product $U_0 \times \ldots \times U_{n-1}$ we will understand the set of functions $f \in \square n \rightarrow \bigcup(i \in \square n \,.\, U_i)$, such that $f\ i \in U_i$. This provides a way to select elements from a tuple. For example:

$\langle a, b, c\rangle\ 0 = a$
$\langle a, b, c\rangle\ 1 = b$
$\langle a, b, c\rangle\ 2 = c$

This definition also gives a meaning to U^1 and U^0. Note that this definition implies that $E^1 \neq E$.

Expressions of the form $\langle a_0, \cdots a_{n-1}\rangle\ i$ are rewritten by the macro expander (to a_i), if i is in the proper range.

An expression a of type U and an expression b of type U^n may be combined into an expression of type U^{n+1} by *cons*ing them: $a : b$. In general, $a_0 : \ldots : a_{n-1} : \langle\rangle$ is equivalent to $\langle a_0, \ldots, a_{n-1}\rangle$.

3.7.5 Parametrized types

The language also contains possibilities to introduce parametrized types, although that parameter is restricted to the domain of the integers. By writing **Int** $\ni n$ in a type one introduces n as a type parameter, which may then be used to impose restrictions on the sizes of the constituents in the rest of the type. Thus one may specify

$$\textbf{Int} \ni n \rightarrow E^n \times E^n \Rightarrow E^n$$

Expansion of a macro of this type yields a system that has two bundles of n wires as input and one bundle of size n as output, where n is determined by the actual macro arguments.

Type checking is a necessary part of the Glass "describing environment". Since it is not decidable (statically) if a type U^n is of exactly the right size (because of the parametrized types), all types of the form U^n are treated in the same way during a first typecheck, namely as types for lists of undetermined size. After macro expansion it is checked if they are of exactly the right size. Since U^n and the n-fold product $U \times \ldots \times U$ are equivalent, the latter type is treated in the same way. Because of this second typecheck it may happen that if you write *or* $\langle a, b, c\rangle$, the fact that there is one input too many will be discovered only in the second typecheck after macro expansion.

3.7.6 Type naming

To save writing in type declarations one may introduce a name for a type:

$$\textbf{type} \quad tp = E \times E \Rightarrow E$$

One may then use this name as a synonym of the righthand side in other declarations, like $my\_xor \in tp$.

3.7.7 Type variables

As is apparent from the *triple macro* in subsection 3.7.1, macro definitions may sometimes be insensitive to the exact form of (part of) their argument. In this example, the macro definition remains perfectly valid whatever the exact form of the parameter A is, as long as its inputs and outputs are of the same type (because these are connected in series). One is therefore allowed to introduce so called *type variables*, thereby introducing polymorphic definitions in Glass. By

$$\{U\}\, V$$

we mean any type V in which U is replaced by an arbitrary type (for a connection). Therefore one is allowed to declare *triple* in the following way:

$$triple \in \{U\}\ U \Rightarrow U \rightarrow U \Rightarrow U$$

This saves defining several variants of *triple*, that differ only in the type. In this way one may write *triple halfadder* and *triple divide_by_two*.

3.8 Macros

3.8.1 A first expansion

When using the Glass "describing environment", it is first checked if a Glass text is correct in syntax and typing. Next the macro expander expands all macro applications. A second type check then checks that all composite terminals are of the proper size.

When the macro expander expands a macro of type $U \rightarrow V$, only the formal arguments of type U are replaced by their corresponding actual arguments (betareduction). As macro expansion results in system descriptions, the result type of a macro is always of the form $U \Rightarrow V$ or $[U]$. Depending on the result type a λ-abstraction or σ-abstraction will be generated. A first example of a macro is the following:

basetype E;

atom *divide_by_two* $\in E \Rightarrow E$;

mac *triple* $\in E \Rightarrow E \rightarrow E \Rightarrow E$;
triple A *in* $= A\ (A\ (A$ *in*$))$;

def *divide by 8* $\in E \Rightarrow E$;
divide by_8 $c =$ *triple divide_by_two* c;

The macro expander will replace *triple divide_by_two* (in *divide_by_8*) by the right hand side of the macro while substituting *divide_by_two* for A, resulting in the following kernel Glass description:

def *divide_by_8* $\in E \Rightarrow E$;
divide_by_8 $c = \lambda in.$(*divide_by_two* (*divide_by_two* (*divide by two in*))) c

Thus *divide_by_8* turns out to be a cascade of three *divide by_two* circuits.

3.8.2 Calculation in macros

As it must be possible to include conditional parts in macros the language has a conditional, whose syntax was adopted from SASL [Tur79]: by

$b \rightarrow e_1; e_2$

is meant that the expansion depends on the condition b; if this condition is true the expander has to expand e_1, in the other case it has to expand e_2. Using these constructs we can give a more general description of a replicated system, which also shows the possibilities of using recursive macros:

basetype E;

atom *divide_by_two* $\in E \Rightarrow E$;

mac *chain* $\in$ **Int** $\rightarrow E \Rightarrow E \rightarrow E \Rightarrow E$;
chain n A *in* $= n = 0 \rightarrow$ *in*; A (*chain* $(n-1)$ A *in*);

def *divide_by_1024* $\in E \Rightarrow E$;
divide_by_1024 *clock* = *chain* 10 *divide_by_two* *clock*;

3.8.3 Using patterns

In order to enable case distinction in a more elegant way, the language contains pattern matching facilities for macros. Macros may consist of several alternatives which are tried from top to bottom at expansion time. The first alternative whose formal arguments match the actual ones is then expanded. Patterns may be:

- a name, matching the actual argument always.
- a constant, matching only if the actual argument equals that constant after its expansion.
- the empty tuple $\langle\rangle$ matching only the empty tuple.
- a tuple $\langle f_1, \ldots, f_n \rangle$, matching of the actual argument has the form $\langle a_1, \ldots, a_n \rangle$ and if f_i matches a_i.
- a consing $a : b$, matching only if the actual argument has the form $c : d$, and a matches c and b matches d.

We must keep in mind, that the actual argument, after expansion, must have precisely the required syntactic form. Using pattern matching (on the integer argument) we are now able to specify an adder chain:

basetype E;

atom *adc* $\in E \times E \times E \Rightarrow E \times E$;

```
mac nbitsadder ∈ Int ∋ n → E^n × E^n × E ⇒ E × E^n;
    nbitsadder 0 (⟨⟩, ⟨⟩, c) = ⟨c, ⟨⟩⟩;
    nbitsadder n ⟨a : as, b : bs, c_in⟩ = ⟨c_out, s : ss⟩
        where
            ⟨v, s⟩ = adc ⟨a, b, c_in⟩;
            ⟨c_out, ss⟩ = nbitsadder (n − 1) ⟨as, bs, v⟩;
        endwhere;

def Fourbitsadder ∈ E^4 × E^4 × E ⇒ E × E^4;
    Fourbitsadder ⟨as, bs, c_in⟩ = nbitsadder 4 ⟨as, bs, c_in⟩;
```

i the expansion (of *Fourbitsadder*) the second alternative of *nbitsadder* is used 4 time or arguments 4, 3, 2 and 1: these do not match the formal argument 0). Finally th :st alternative is used (because the actual and formal argument are both 0).

.8.4 A larger example

n illustrative example of the application of (local) macros is the following descriptio ' a four-bit counter:

```
atom jkff ∈ E × E ⇒ E,
     and2 ∈ E × E ⇒ E,
     and3 ∈ E^3 ⇒ E,
     and4 ∈ E^4 ⇒ E,
     and5 ∈ E^5 ⇒ E,
     or ∈ E × E ⇒ E,
     buf ∈ E ⇒ E,
     nand ∈ E × E ⇒ E;

def counter ∈ E^4 × E × E × E × E ⇒ E^4 × E;
    counter⟨⟨d0, d1, d2, d3⟩, nload, nclear, enap, enat⟩ =
        ⟨⟨q0, q1, q2, q3⟩, carry⟩
        where
          q0 = counterstage ⟨d0, preset, en, nclr⟩;
          q1 = counterstage ⟨d1, preset, and2 ⟨en, q0⟩, nclr⟩;
          q2 = counterstage ⟨d2, preset, and3 ⟨en, q0, q1⟩, nclr⟩;
          q3 = counterstage ⟨d3, preset, and4 ⟨en, q0, q1, q2⟩, nclr⟩;
          carry = and5 ⟨enat, q0, q1, q2, q3⟩;
          preset = nand ⟨nload, nclr⟩;
```

```
Mac counterstage ∈ E⁴ ⇒ E;
      counterstage ⟨d, preset, toggle, nclr⟩ =
         jkff ⟨and2⟨ja, ea⟩, and2 ⟨ka, ea⟩⟩
              where
                   ja = nand ⟨ka, preset⟩;
                   ka = nand ⟨d, nclr⟩;
                   ea = or ⟨toggle, preset⟩;
              endwhere;
endwhere;
```

whose structural interpretation can be found in figure 3.1. Note that the **where**-clause is now used for two purposes: to give names to outputs and to introduce local macros.

3.8.5 A checkerboard example

Another example of the use of macros is the following description of a checkerboard circuit:

```
atom  odd ∈ E × E ⇒ E × E,
      even ∈ E × E ⇒ E × E;

mac  row ∈ Int → Int ∋ l → E^l × E ⇒ E^l × E;
     row k 0 ⟨⟨⟩, e⟩ = ⟨⟨⟩, e⟩;
     row k l ⟨n : N, e⟩ = ⟨s : S, w⟩
          where
               ⟨s, ie⟩ = (k = 0 → even; odd) ⟨n, e⟩;
               ⟨S, w⟩ = row (1 − k) (l − 1) ⟨N, ie⟩;
          endwhere;

mac  board ∈ Int → Int ∋ l → Int ∋ m → E^l × E^m ⇒ E^l × E^m;
     board k l 0 ⟨N, ⟨⟩⟩ = ⟨N, ⟨⟩⟩;
     board k l m ⟨N, e : E⟩ = ⟨S, w : W⟩
          where
               ⟨IS, w⟩ = row k l ⟨N, e⟩;
               ⟨S, W⟩ = board (1 − k) l (m − 1) ⟨IS, E⟩;
          endwhere;

def  chess ∈ E⁸ × E⁸ ⇒ E⁸ × E⁸;
     chess ⟨N, E⟩ = board 0 8 8 ⟨N, E⟩

def  smallchess ∈ E⁴ × E⁴ ⇒ E⁴ × E⁴;
     smallchess ⟨N, E⟩ = board 0 4 4 ⟨N, E⟩
```

whose structural interpretation can be found in figure 3.2.

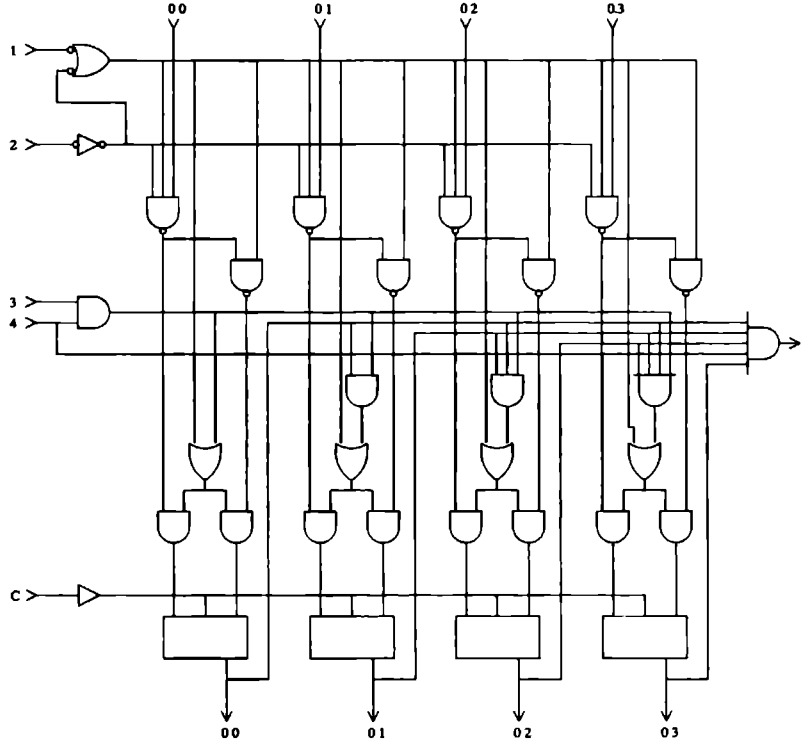


Figure 3.1: Structural interpretation of counter description

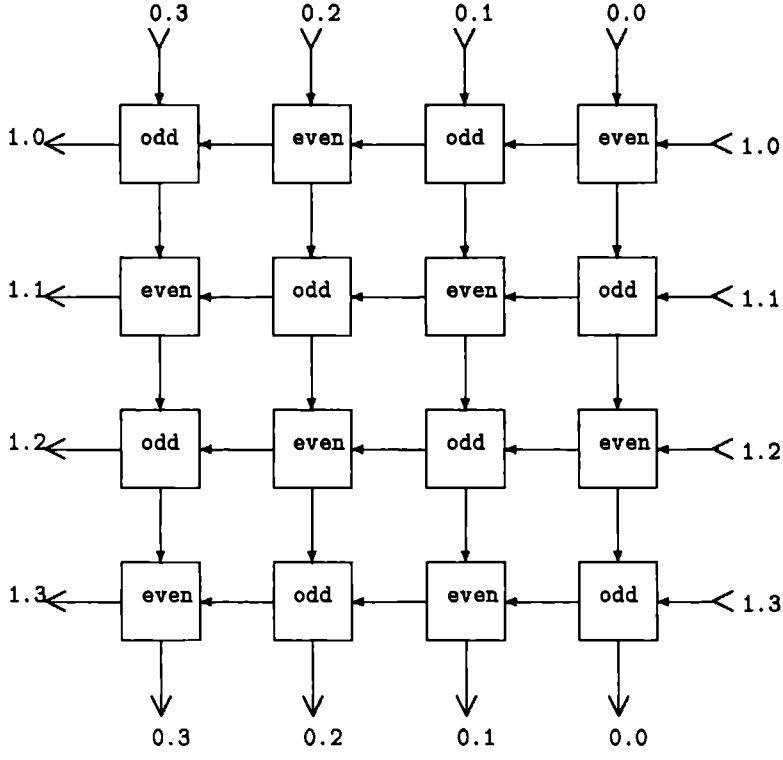


Figure 3.2: Structural interpretation of chessboard description

.9 From Glass to Reals

lass has been implemented succesfully in Esprit project Forfun. In the near futu lass will be embedded in Funmath as the subset Reals. As such the syntax of t ystem description language will be changed to fit in the Funmath framework. Th ill improve the clarity of system descriptions. A reason for this is that Glass as recursor to Funmath was not yet as orthogonal as Funmath.

The embedding will have the advantage that only one formalism will be use r describing systems and for describing semantic functions. Moreover some of t mitations, currently present in Glass (such as the impossibility of describing arbitra terconnection patterns in an elegant way because general indexing is not allowed i ft hand sides of local definitions or the absence of quantification of indexes) can b iminated in this process.

Uncommitted types are added to Funmath in the following way. A fourth gener pe is introduced namely $\mathcal{E}$, the type of all basetypes. Basetypes and atomic definitio e then introduced by definitions without a **with** part:

def $E : \mathcal{E}$

def $and : E^2 \Rightarrow E$
$or : E^2 \Rightarrow E$
$not : E \Rightarrow E$

ystem definitions can then be introduced in the usual Funmath syntax:

def $select : E^3 \Rightarrow E$
with $select\ (s, a, b) = or\ (and\ (not\ s, a),\ and\ (s, b))$

Macros will also be added in the next version of Funmath. By allowing gener dexing a powerful system description language will emerge. Consider for instan e following description in Reals :

mac $shuffle : \mathbf{Z} \ni n \rightarrow E^n \Rightarrow E$
$shuffle\ n\ in\ i = in\ (n - i)$

Chapter 4

Simple semantic functions

In Dwimordene, in Lorien
Seldom have walked the feet of Men,
Few mortal eyes have seen the light
That lies there ever, long and bright.
Galadriel! Galadriel!
Clear is the water of your well;
White is the star in your white hand;
Unmarred, unstained is leaf and land
In Dwimordene, in Lorien
More fair than thoughts of Mortal Men.
Lord of the Rings *J.R.R. Tolkie*

.1 Introduction

the following chapters we will investigate the definition and implementation of s
antic functions. As we already explained in a previous chapter, semantic functio
ap descriptions written in Glass (i.e. Glass syntax) onto descriptions in particular d
ains of interest. As syntax is very important in describing these semantic function
e will introduce a specific notation to separate Glass syntax from the notation use
the metalevel for describing semantic functions. The two special brackets '**[**' an
' will be placed around Glass syntax or Glass syntactic constructs in the tradition
enotational semantics.

Semantic functions may be divided in two classes, namely the compositional on
id the non-compositional ones. Compositional semantic functions express the pro
ties of compound system definitions in terms of the properties of the subsystem
escriptions of subsystems that are not compound but primitive (atomic systems) a
apped by the semantic function onto their predefined semantics. Non-composition
mantic functions are not expressible in this convenient fashion, in the sense th
ie meaning of a composite cannot be expressed in terms of the meanings of the co

stituents, but involves additional information.

We will describe the semantic functions initially as mappings from Glass syntax to Funmath. The domain of interest will therefore also be described in Funmath. Later on we will also indicate how one transforms these mappings into mappings from Glass syntax to programs in existing programming languages. Of these latter we will consider especially mappings towards the functional programming language Miranda and the imperative programming language C.

The first of these is interesting because one can do a one-to-one translation from the mapping towards the computational subset of Funmath, Comma, into the mapping towards Miranda. The second is interesting because much efficiency can be gained if one can map into an imperative language, due to the fact that really efficient implementations of lazy functional languages are still lacking.

We will now focus on compositional semantic functions for directional system descriptions. In general they all conform to the following generic model, presented in [Bou88].

4.2 The generic model for the directional subset

One can express a semantic function by giving rules for every syntactic construct in the directional subpart of Glass indicating how properties are determined by those of the constituting parts. In fact, in this section we define the specification of a transducer from Glass syntax to Funmath syntax. The semantic functions defined in the following sections and some chapters all adhere to these rules. The only difference between them lies in the definition of the properties of the atoms.

In this generic model we will consider for the time being only a small subset of Glass, namely the constructs for directional system descriptions without local definitions or system parameters. In the next chapter we will alleviate this restriction gradually.

Our transducer will map different syntactic constructs in different ways. For every different syntactic construct a different function must be defined. To distinguish between the various functions, we will add the appropriate syntactic construct as an index to the function.

Not only the body of a system description must be mapped by a semantic function but also its type. Let $\mathcal{D}$ be the domain of interpretation and E be the base type used in the Glass descriptions. We define:

$$
\begin{array}{l}
\textbf{def}\ \ \tau_T : \mathcal{T}_{DGl} \rightarrow \mathcal{T} \\
\textbf{with}\ \tau_T\,[\![E]\!] = \mathcal{D} \\
\qquad \tau_T\,[\![T_0\ \&\ \ldots\ \&\ T_{n-1}]\!] = \tau_T\,[\![T_0]\!] \times \ldots \times \tau_T\,[\![T_{n-1}]\!] \\
\qquad \tau_T\,[\![U \Rightarrow V]\!] = \tau_T\,[\![U]\!] \rightarrow \tau_T\,[\![V]\!]
\end{array}
$$

where $\mathcal{T}_{DGl}$ is the collection of all directional types in Glass. Let $\mathcal{A}$ be the set of atoms. The interpretation of the atoms then takes as type:

$$\tau_A : \mathcal{A} \to \{\tau_T\}$$

To be more specific, for an atom $c : \mathcal{A}$

$$\tau_A \, [\![c]\!] \in \tau_T \, [\![typeof \; c]\!]$$

In the following rules $\mathcal{E}_{Gl}$ and $\mathcal{F}_{Gl}$ stand for the set of (syntactic) correct Glass expressions and system definitions, respectively. The set $\mathcal{I}$ is a set of mappings from Glass variables to Funmath variables. In the following description $i \in \mathcal{I}$ is used to carry the environment in the transduction.

$$\begin{array}{ll}
\textbf{def} & \tau_E : \mathcal{E}_{Gl} \to \mathcal{I} \to \{\tau_T\} \\
\textbf{with} & \tau_E \, [\![v]\!] \, i = i \, v \qquad\qquad \text{if } v \text{ is a variable} \\
& \tau_E \, [\![f \; e]\!] \, i = \tau_F \, [\![f]\!] \, i \;\; (\tau_E \, [\![e]\!] \, i) \\
& \tau_E \, [\![\langle e_0, e_1, \ldots, e_{n-1} \rangle]\!] \, i = (\tau_E \, [\![e_0]\!] \, i, \ldots, (\tau_E \, [\![e_{n-1}]\!] \, i)
\end{array}$$

$$\begin{array}{ll}
\textbf{def} & \tau_F : \mathcal{F}_{Gl} \to \mathcal{I} \to \{\tau_T\} \\
\textbf{with} & \tau_F \, [\![c]\!] \, i = \tau_A \, [\![c]\!] \qquad\qquad \text{if } c \in \mathcal{A} \\
& \tau_F \, [\![\lambda \; t.e]\!] \, i : \tau_T \, [\![typeof \; (\lambda \; t.e)]\!] \qquad \text{with} \\
& \tau_F \, [\![\lambda \; t.e]\!] \, i \; d = \tau_E \, [\![e]\!] \;\; i[\,{}^{d}/_{t}] \\
& \tau_F \, [\![\lambda \; \langle t_0, t_1, \ldots, t_{n-1} \rangle.e]\!] \, i : \tau_T \, [\![typeof \; (\lambda \; \langle t_0, t_1, \ldots, t_{n-1} \rangle.e)]\!] \qquad \text{with} \\
& \tau_F \, [\![\lambda \; \langle t_0, t_1, \ldots, t_{n-1} \rangle.e]\!] \, i \; (d_0, d_1, \ldots, d_{n-1}) = \\
& \qquad \tau_E \, [\![e]\!] \;\; i[\,{}^{d_0}/_{t_0}] \ldots [\,{}^{d_{n-1}}/_{t_{n-1}}]
\end{array}$$

where we use the following function to auxiliary functional mapping (with postfix notation) mapping a given function onto a function that differs from the given function for only one value in the domain:

$$\begin{array}{ll}
\textbf{def} & - [\,{}^{d}/_{t}] : \mathcal{I} \to \mathcal{I} \\
\textbf{with} & i \, [\,{}^{d}/_{t}] \; t' = \; (t' = t)?d\natural i \; t'
\end{array}$$

Remark that we differentiate in the transduction of a system definition having only one formal parameter and the transduction of one having a tuple of formal parameters. We must do so because in Glass these are two different syntactical forms. The reason for this is convenience for the user as well as for the writer of semantic functions: single variables of base type E need not be encapsulated as 1-tuples of type E^1.

The meaning of a specific system definition may then be found as follows:

$$\begin{array}{ll}
\textbf{def} & \tau_S : \mathcal{F}_{Gl} \to \{\tau_T\} \\
\textbf{with} & \tau_S \, [\![s]\!] = \tau_F \, [\![s]\!] \, i \qquad \text{where } i \text{ may be any interpretation}
\end{array}$$

4.3 The simplex model

One of the simple semantic functions is the semantic function *simplex*, which yields the statical semantics of a description. As domain of interest we take the set $\mathbf{B} = \{0, 1\}$.

For this particular semantic function we will extensively show how it corresponds to the generic model. However, we will not repeat this exercise for the other semantic functions presented in other sections of this chapter, since this can be done in a quite similar and straightforward way.

First we introduce the interpretations of our atoms in our domain of interest:

def $s\_not : \mathbb{B} \to \mathbb{B}$ **with** $s\_not\ c = \neg c$
def $s\_and : \mathbb{B}^2 \to \mathbb{B}$ **with** $s\_and\ (a, b) = a \wedge b$
def $s\_nand : \mathbb{B}^2 \to \mathbb{B}$ **with** $s\_nand\ (a, b) = \neg\ (a \wedge b)$
def $s\_or : \mathbb{B}^2 \to \mathbb{B}$ **with** $s\_or\ (a, b) = a \vee b$
def $s\_nor : \mathbb{B}^2 \to \mathbb{B}$ **with** $s\_nor\ (a, b) = \neg(a \vee b)$
def $s\_xor : \mathbb{B}^2 \to \mathbb{B}$ **with** $s\_xor\ (a, b) = a \oplus b$
def $s\_xnor : \mathbb{B}^2 \to \mathbb{B}$ **with** $s\_xnor\ (a.b) = a \equiv b$

We then define:

$\tau_A\ [\![not]\!] = s\ not$
$\tau_A\ [\![and]\!] = s\_and$
$\tau_A\ [\![nand]\!] = s\_nand$
$\tau_A\ [\![or]\!] = s\_or$
$\tau_A\ [\![nor]\!] = s\_nor$
$\tau_A\ [\![xor]\!] = s\_xor$
$\tau_A\ [\![xnor]\!] = s\_xnor$

Elaboration of the resulting function is illustrated for the following description of a simple selector:

def $hsel \in E \times E \times E \Rightarrow E;$
$hsel\ \langle s, x_0, x_1\rangle = or\ \langle and\ \langle not\ s, x_0\rangle, and\ \langle s, x_1\rangle\rangle;$

The meaning of this selector description can be derived by application of the generic model:

$(simplex\ [\![hsel]\!]) : \mathbb{B}^3 \to \mathbb{B}$
$(simplex\ [\![hsel]\!])\ (d, d_0, d_1) =$
$\quad \tau_F\ [\![\lambda(s, x_0, x_1).or\ (and\ (not\ s, x_0), and\ (s, x_1))]\!]\ i\ \ (d, d_0, d_1) =$
$\quad \tau_E\ [\![or\ (and\ (not\ s, x_0), and\ (s, x_1))]\!]\ i[\,^{d}/_{s}][\,^{d_0}/_{x_0}][\,^{d_1}/_{x_1}] =$
$\quad \tau_E\ [\![or\ (and\ (not\ s, x_0), and\ (s, x_1))]\!]\ i' =$
$\quad \tau_F\ [\![or]\!]\ i'\ \ (\tau_E\ [\![and\ (not\ s, x_0)]\!]\ i', \tau_E\ [\![and\ (s, x_1)]\!]\ i') =$
$\quad \tau_A\ [\![or]\!]\ \ (\tau_E\ [\![and\ (not\ s, x_0)]\!]\ i', \tau_E\ [\![and\ (s, x_1)]\!]\ i') =$
$\quad s\_or\ (\tau_E\ [\![and\ (not\ s, x_0)]\!]\ i', \tau_E\ [\![and\ (s, x_1)]\!]\ i') =$
$\quad s\_or\ (\tau_F\ [\![and]\!]i'\ \ (\tau_E\ [\![not\ s]\!]\ i', \tau_E\ [\![x_0]\!]\ i'), \tau_F\ [\![and]\!]i'\ \ (\tau_E\ [\![s]\!]\ i', \tau_E\ [\![x_1]\!]\ i')) =$
$\quad s\_or\ (\tau_A\ [\![and]\!]\ \ (\tau_E\ [\![not\ s]\!]\ i', \tau_E\ [\![x_0]\!]\ i'), \tau_A\ [\![and]\!]\ \ (\tau_E\ [\![s]\!]\ i', \tau_E\ [\![x_1]\!]\ i')) =$
$\quad s\_or\ (s\ and\ (\tau_E\ [\![not\ s]\!]\ i', \tau_E\ [\![x_0]\!]\ i'), s\_and\ (\tau_E\ [\![s]\!]\ i', \tau_E\ [\![x_1]\!]\ i')) =$
$\quad s\_or\ (s\_and\ (\tau_F\ [\![not]\!]\ i'\ (\tau_E\ [\![s]\!]\ i'), d_0), s\_and\ (d, d_1)) =$

$$s\_or\ (s\ and\ (\tau_A\ [\![not]\!]\ (\tau_E\ [\![s]\!]\ i'), d_0), s\_and\ (d, d_1)) =$$
$$s\_or\ (s\ and\ (s\_not\ (\tau_E\ [\![s]\!]\ i'), d_0), s\_and\ (d, d_1)) =$$
$$s\_or\ (s\_and\ (s\_not\ d, d_0), s\_and\ (d, d_1))$$

where we have substituted

$$i' : \mathcal{I}$$
$$i' = i\ [\ ^{d}/_{s}][\ ^{d_0}/_{x_0}][\ ^{d_1}/_{x_1}]$$

We can use this result to study the static behaviour of this system. More specifically:

$$(simplex\ [\![hsel]\!])\ \ (d, d_0, d_1) = (\neg d \wedge d_0) \vee (d \wedge d_1)$$

One can easily implement a semantic function which will yield corresponding executable definitions in a programming language like C or Miranda.

4.4 Eichelberger algebra

Another semantic function is a variant on the semantic function *simplex*, based on the Eichelberger algebra [Eic65]. We will denote this semantic function by *eich*.

Our domain of interpretation in this model is the set $\mathbb{E} = \{0, \perp, 1\}$, where $\perp$ stands for "undefined value". As previously, we must first specify the properties of our atoms:

def $e\_not : \mathbb{E} \to \mathbb{E}$ $\qquad\qquad \tau_A\ [\![not]\!] = e\_not$
with $e\_not\ a = a = \perp ?\perp \dagger s\_not\ a$

def $e\ and : \mathbb{E}^2 \to \mathbb{E}$ $\qquad\qquad \tau_A\ [\![and]\!] = e\_and$
with $e\_and\ (a, b) = (a = 1) ?b \dagger a = 0 ?0 \dagger b = 0 ?0 \dagger \perp$

def $e\_or : \mathbb{E}^2 \to \mathbb{E}$ $\qquad\qquad \tau_A\ [\![or]\!] = e\ or$
with $e\_or\ (a, b) = (a = 0) ?b \dagger a = 1 ?1 \dagger b = 1 ?1 \dagger \perp$

def $e\_or3 : \mathbb{E}^3 \to \mathbb{E}$ $\qquad\qquad \tau_A\ [\![or3]\!] = e\_or3$
with $e\_or3\ (a, b, c) = e\_or\ (a, e\_or\ (b, c))$

Let us consider a slightly changed version of the selector of the previous section:

def $ssel \in E \times E \times E \Rightarrow E;$
$\quad ssel\ \langle s, x_0, x_1 \rangle = or3\ \langle and\ \langle not\ s, x_0 \rangle, and\ \langle s, x_1 \rangle, and\ \langle x_0, x_1 \rangle \rangle$

This semantic function also conforms to our generic model. If we apply *eich* to our selectors *hsel* and *ssel* one deduces that:

$$(eich\ [\![hsel]\!]) : \mathbf{E}^3 \to \mathbf{E}$$
$$(eich\ [\![hsel]\!])\ (d, d_0, d_1) = e\_or\ (e\_and\ (e\_not\ d, d_0), e\_and\ (d, d_1))$$

$$(eich\ [\![ssel]\!]) : \mathbf{E}^3 \to \mathbf{E}$$
$$(eich\ [\![ssel]\!])\ (d, d_0, d_1) = e\_or3\ (e\_and\ (e\_not\ d, d_0), e\_and\ (d, d_1), e\_and\ (d_0, d_1))$$

From this we can derive that:

$$\begin{aligned}&(eich\ [\![hsel]\!])\ (\bot, 1, 1) = \\ &\quad e\_or\ (e\_and\ (e\_not\ \bot, 1), e\_and\ (\bot, 1)) = \\ &\quad e\_or\ (\bot, \bot) = \\ &\quad \bot\end{aligned}$$

and that:

$$\begin{aligned}&(eich\ [\![ssel]\!])\ (\bot, 1, 1) = \\ &\quad e\_or3\ (e\_and\ (e\_not\ \bot, 1), e\_and\ (\bot, 1), e\_and\ (1, 1)) = \\ &\quad e\_or3\ (\bot, \bot, 1) = \\ &\quad 1\end{aligned}$$

Thus we have shown that $eich\ [\![hsel]\!] \neq eich\ [\![ssel]\!]$, although it can be shown using ordinary boolean algebra that $simplex\ [\![hsel]\!] = simplex\ [\![ssel]\!]$. This observation is related to the fact that *ssel* is safe with respect to static hazards and *hsel* is not.

4.5 Semantic functions and tristate logic

We can define a semantic function *tsimp* that handles tristate logic by introducing the following domain of interpretation

$$\mathcal{D}_T = \{Z, 0, 1, E\}$$

where Z denotes high impedance and E an erroneous condition (occurring for instance when two tristate buffers are attempting to drive a common output to two different levels). On this domain the following lattice is defined:

$$\begin{array}{ll} Z \sqsubseteq 0 & 0 \sqsubseteq E \\ Z \sqsubseteq 1 & 1 \sqsubseteq E \end{array}$$

This lattice induces the following higher order operator (see also chapter 1) and binary counterpart:

$$\begin{aligned}&\textbf{def}\ \ \bigsqcup : Fcod\ \mathcal{D}_T \to \mathcal{D}_T \\ &\textbf{with}\ \forall(a : \mathcal{D}_T \bullet \bigsqcup f \leq a \Leftrightarrow \forall(x : \{f\} \bullet x \leq a))\end{aligned}$$

$$\begin{aligned}&\textbf{def}\ \ \sqcup : \mathcal{D}_T \leftrightarrow \mathcal{D}_T \to \mathcal{D}_T \\ &\textbf{with}\ a \sqcup b = \bigsqcup (a, b)\end{aligned}$$

Clearly:

$$\forall(a : \mathcal{D}_T \cdot a \sqcup Z = a \wedge a \sqcup E = E)$$

We then define the properties of the atoms, declared in subsection 3.4.5.

def $t\_not : \mathcal{D}_T \to \mathcal{D}_T$
with $t\_not\ a = (a = 0)?1 \dagger (a = 1)?0 \dagger E$

def $t\_nand : \mathcal{D}_T^2 \to \mathcal{D}_T$
with $t\_nand\ (a, b) = (a = 0)?1 \dagger (b = 0)?1 \dagger (a = 1 \wedge b = 1)?0 \dagger E$

def $t\_tbuf : \mathcal{D}_T^2 \to \mathcal{D}_T$
with $t\_tbuf\ (en, a) = (en = 1)?Z \dagger (en = 0 \wedge a \in \{0, 1\})?a \dagger E$

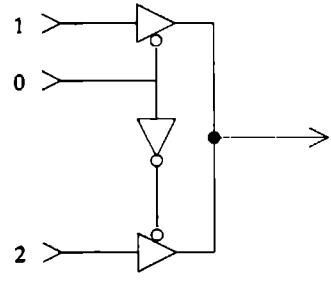
Also:

$\tau_A\ [\![not]\!] = t\ not$
$\tau_A\ [\![nand]\!] = t\_nand$
$\tau_A\ [\![tbuf]\!] = t\_tbuf$
$\tau_A\ [\![join_2]\!] = \bigsqcup$
$\tau_A\ [\![join_3]\!] = \bigsqcup$
$\tau_A\ [\![join_4]\!] = \bigsqcup$

Consider the following description of a bus driven from two sources:

def $bus_2 \in E \times E \times E \Rightarrow E$
$bus_2\ \langle en, a, b\rangle = join_2\ \langle tbuf\ \langle en, a\rangle, tbuf\ \langle not\ en, b\rangle\rangle$

which has the following structural interpretation:



The semantic function *tsimp* conforms to the generic model. We therefore immediately derive:

$tsimp\ [\![bus_2]\!] : \mathcal{D}_T \times \mathcal{D}_T \times \mathcal{D}_T \to \mathcal{D}_T$
$tsimp\ [\![bus_2]\!]\ (en, a, b) = \bigsqcup (t\_tbuf\ (en, a), t\ \ tbuf\ (t\_not\ en, b))$

Consequently:

$$
\begin{array}{ll}
 & tsimp\ [\![bus_2]\!]\ (0,a,b) \\
= \{\text{def } tsimp\} & \bigsqcup (t\_tbuf\ (0,a), t\_tbuf\ (t\_not\ 0,b)) \\
= \{\text{def } t\_tbuf, t\_not\} & \bigsqcup (a \in \{0,1\}?a\dagger E, Z \\
= \{\text{prop } \bigsqcup\} & a \in \{0,1\}?a\dagger E
\end{array}
$$

Likewise

$$
\begin{array}{l}
tsimp\ [\![bus_2]\!]\ (1,a,b) = b \in \{0,1\}?b\dagger E \\
tsimp\ [\![bus_2]\!]\ (Z,a,b) = E \\
tsimp\ [\![bus_2]\!]\ (E,a,b) = E
\end{array}
$$

4.6 Some very simple cost models

When designing hardware it is advisable to have some measure of the cost of the resulting circuit. If our design will be realized in VLSI the cost of the circuit of the design must be measured by the chip area needed for the circuit. To calculate an estimate of this area one must place the subsystems of a circuit on the silicon in an optimal way. This latter is unfortunately a highly non-compositional and very time consuming process.

There are also other models with which we may obtain some measure of the cost of a design. We may consider for instance the number of gates in a design. These kind of models do comply to a compositional model although they do not comply to the generic model.

As stated when we defined the generic model we must provide rules for every syntactic construct. Define:

def $\tau_A : \mathcal{A} \to \mathbb{R}$
with $\tau_A\ [\![c]\!]$ = the cost of the atom c

Likewise we define the other rules:

def $\tau_E : \mathcal{E}_{GI} \to \mathbb{R}$
with $\tau_E\ [\![v]\!] = 0$ if v is a variable
$\tau_E\ [\![f\ e]\!] = \tau_F\ [\![f]\!] + \tau_E\ [\![e]\!]$
$\tau_E\ [\![\langle e_0, \ldots, e_{n-1}\rangle]\!] = \sum(i : \Box n \bullet \tau_E\ [\![e_i]\!])$

def $\tau_F : \mathcal{F}_{GI} \to \mathbb{R}$
with $\tau_F\ [\![c]\!] = \tau_A\ [\![c]\!]$ if $c \in \mathcal{A}$
$\tau_F\ [\![\lambda t.e]\!] = \tau_E\ [\![e]\!]$
$\tau_F\ [\![\lambda\langle t_0, \ldots, t_{n-1}\rangle.e]\!] = \tau_E\ [\![e]\!]$

If we are interested in the number of gates in a circuit it suffices to define $\tau_A\ [\![c]\!] = 1$ for every gate c. A better cost estimate is obtained by associating with each atom its appropriate cost, in which case the above rules will yield the sum of the costs of all gates in the circuit.

A special case of this can be defined when using the number of inputs as the cos
'the gate. It is this cost model that forms the basis of the optimizing processes usin
arnaugh diagrams or Quine- McCluskey techniques which we may use to reduce th
ımber of literals (i.e. the number of inputs in the circuit) in a sum of products.

.7 Conclusions

generic model for compositional semantic functions for the directional subset of Gla
presented. Three different static behavioural models conforming to this generic mod
ıve been discussed.

Behavioural descriptions obtained by application of these semantic functions di
ɔt involve higher order functions (or only one whose application can easily be tran
rmed into one not involving higher order functions), which means that they can easil
ɜ transformed into executable descriptions in an imperative programming languag
ıch as C.

Two very simple cost models were also presented, that are also easy to implemen

Chapter 5

Implementation aspects

There hammer on the anvil smote
There chisel clove, and graver wrote,
There forged was blade, and bound was hilt,
The delver mined, the mason built
There beryl, pearl, and opale pale,
And metal wrought like fishes' mail,
Buckler and corslet, axe and sword,
And shining spears were led in hoard
Lord of the Rings J R R Tolkien

5.1 History of the Forfun environment

In December 1985 Esprit project proposal 881 [Bou85] was submitted to the European Commission to design and implement a prototype system description language, based on the principle of system semantics (evolving into the language Glass during the course of the project) and a prototype system description environment based upon this language, thus demonstrating the feasibility of the principle of system semantics as well as demonstrating the feasibility of writing semantic functions Its title was "Formal description of arbitrary systems by means of functional formalisms", also known by its acronym "Forfun"

The proposal was accepted, enabling the project to start in June 1986 Four partners were involved, namely the Catholic University of Nijmegen (prime contractor), Sagantec BV , the Technical University of Delft and Bell Telephone Manufacturing Company in Antwerpen

It was soon apparent that the implementation could be split into two parts

a) A part common to all interpretations of Glass, consisting of a parser, type checker and macro expander Later experimentation showed that the type checker had to be split up in two parts one which does the initial type check and one that type checks the output of the macro expander This part of the environment was

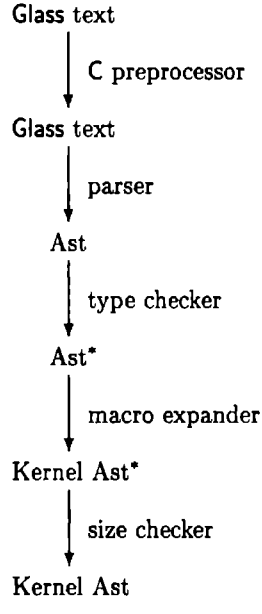
called the *front end* of the environment. Earlier versions of the environment also contained programs supporting (precompiled) design libraries. These were later omitted to simplify the environment.

b) The set of semantic functions which only have to work on an internal representation that represents only the subset of the language without macros, the socalled *kernel* language.

In June 1988 a preliminary version of the environment had been implemented which consisted of a parser (written in Yacc and C) and a typechecker (written in Pascal) and several small semantic functions. What remained to be implemented was the macro expander and an extensive set of semantic functions.

5.2 The current describing environment

As we have already seen the describing environment consists of two parts. The front end translates from Glass into a kernel Glass abstract syntax tree (Ast). This translation process takes five passes:



The first pass is formed by the C preprocessor which is used mainly to include other Glass files in the description of a system. In this way some support is still given for component libraries. The second pass parses the syntax of the description and yields

the corresponding abstract syntax tree. The third pass checks the type of every construct in the description and rewrites the abstract syntax tree, distinguishing between system and connection parameters, using the typing information. The fourth pass is formed by macro expansion of all macro applications as well as macro like constructs such as conditionals. The fifth pass is again a type checker which does a much tighter type check than the third pass. This is done because the exact multiplicity of the connections are known only after macro expansion. The output of the fifth pass is an abstract syntax tree of the kernel Glass description of the initial system. After the fifth pass an unparser is provided to generate a Glass file from the generated kernel Glass abstract syntax tree. In this way a user can inspect this file to see whether the resulting kernel Glass description corresponds to what he wished to describe in the full language using macros.

The second part of the environment is formed by a set of semantic functions both for the analog and the digital domain, as well as a set of convenient support libraries.

The abstract syntax tree representations between the passes are based upon a text representation of Miranda like abstract data types [Tur85]. One of the problems in the implementation was that different programming languages were used for different parts of the implementation. The parser and macro expander were written in Glammar [Vos91], which is a subset of Eag [Wat74], [Mei86], whereas the typecheckers were written in Pascal and later rewritten in C. Moreover, the internal representation of the syntax trees evolved with time, necessatating adaptation of the software. Lastly, the various partners of the implementation project used different machines and different variants of the operating system Unix.

5.3 The Forfun directory

In the project many software packages had to be written, which needed to interface with each other. For this reason it was prescribed that the describing environment was mapped on a Unix directory containing subdirectories for every software package. Apart from these one needed also subdirectories named `include`, `lib` and `bin` to install include files, libraries and compiled programs.

In every subdirectory a makefile had to be present specifying at least the following target:

setup: This target is used to modify part of the makefile using local configuration files present in the top directory, thus adapting the makefile to the local situation (search paths, flags, etc.). Remark that this solved the third problem mentioned in the previous section.

all: This target is used to compile the software in the package.

test: This target is used to test the compiled software.

install: This target is used to install the software in the appropriate directories.

clean: This target is used to remove the files that are no longer necessary after installation.

lint: This target is used to perform a type check of the software in the package.

Thus a typical installation run of a software package is performed by entering the sequence of commands:

```
make setup
make all test lint
make install clean
```

5.4 Using Tm

As mentioned earlier the abstract syntax trees yielded by various parts of the environment were based on a text representation of Miranda like abstract data types. Since programs written in different programming languages had to read and write these data structures a way was sought to generate these parts of the programs automatically. A solution was provided by the development of a program called Tm (Template manager) by C. van Reeuwijk.

Tm can generate datastructure definitions, memory management routines, and transput routines from a single datastructure definition file (using two template files provided per supported language). For handling transput the programmer has to provide some support concerning the transput of the primitive types (int, float, string and symbol). Most of this support is given by a support library also implemented by C. van Reeuwijk.

The format of the datastructure definition file is quite similar to the description in Miranda of abstract data types. A declaration in this description format either introduces an abstract data type or a synonym for another type. The format of an abstract datatype declaration is as follows:

$$
\begin{aligned}
\mathit{typename} ::= {} & \mathit{Constructor}_0\ \mathit{field}_{00} \ \ldots\ \mathit{field}_{0n_0} \\
| {} & \mathit{Constructor}_1\ \mathit{field}_{10} \ \ldots\ \mathit{field}_{1n_1} \\
& \vdots \\
| {} & \mathit{Constructor}_m\ \mathit{field}_{m0} \ \ldots\ \mathit{field}_{mn_m}
\end{aligned}
$$

where a field is of the form:

fieldname : *type*

This is a difference with Miranda abstract data types. In Miranda only the type of the field must be given. In our datastructures the field names must also be provided since they will be used as a selector names in the generated C or Pascal datastructure type definitions.

A synonym type is introduced by the following syntax:

typename == *type*

Four kinds of types may be used in the datastructure definition file:

- Primitive types: bool, inum, fnum, string, and symbol.
- Abstract data types identified by their name.
- Sequence types indicated by square brackets.
- Product types indicated by a list of types.

An example description could be the following introducing an abstract syntax tree for some graphical language:

```
|| File: graph.ds
point == (x:inum,y:inum);

element ::= Point p:point
          | Line p1:point p2:point
          | Polygon pp:[point];
```

5.5 Using Tm in C

Since every semantic function implemented by the author was written in the programming language C, we will focus on the support given by Tm for implementing semantic functions in C.

Tm will represent the primitive types by their C counterparts, product types by pointers to structures containing the appropriate fields, abstract data types by pointers to unions prepended by a tag field (similar to the Pascal variant record) and sequence types by pointers to structures pointing to arrays. (These structures are used for memory management purposes). To these types the same names are given as specified in the datastructure definition file.

The generated routines fall in the following six categories. In the description <typ> stands for any of the types specified in the datastructure definition file and <typ>_list for any sequence of the specified type:

1) Memory allocation routines.

   - new_<Constructor>
     This routine allocates a node which has <Constructor> as tag. The other fields are copied from the routine arguments.
   - new_<typ>
     This routine allocates a node of type <typ>.

- new_<typ>_list
  This routine allocates a node of type <typ>_list with 0 elements.
- room_<typ>_list
  This routine allocates memory for a number of elements in a list.

2) Duplication routines.

- rdup_<typ>, rdup_<typ>_list
  These routines duplicate their argument recursively. This is often necessary when transforming datastructures to avoid sharing of datastructures.

3) Freeing routines.

- rfre_<typ>, rfre_<typ>_list
  These routines free their arguments recursively thus making the storage occupied by them available for reuse. To speed up operations a free list is maintained for every type in the description file.

4) List manipulation routines.

- app_<typ>_list, ins_<typ>_list, del_<typ>_list
  These routines provide support for manipulation of lists (sequence types) (to append elements, insert elements at a certain position and delete elements from a position).

5) Output routines.

- fprint_<typ>, fprint_<typ>_list
  These routines will write their argument to a file formatted as an instance of the datastructure definition file.

6) Input routines.

- fscan_<typ>, fscan_<typ>_list
  These routines will read an instance of the definition file while building the corresponding C datastructures.

To obtain these definitions the semantic function writer has to write a file specifying which of the above routines must be generated. Two (standard) template files must also be provided, as well as the datastructure definition file. Usually, for the latter the standard kernel Glass datastructure definition file present in one of the subdirectories of the environment is taken.

5.6 An example

The definitions generated by Tm are very easy to handle. Consider for instance the following piece of code, which implements the cost model based on the number of gates in a hardware design (not all code is shown):

```
def_list all_defs;

int cost_def (d)
 def d;
        { return (cost_val (d -> Valas.vals));
        };

int cost_val (v)
 val v;
        { switch (v -> tag)
            { case TagVSym: { def d = find_def (v -> VSym.sym);
                              if (d != def_nil) return (cost_def (d));
                              return (0);
                            };
              case TagVLambda: return (cost_val (v -> VLambda.lval));
              case TAGVApply: return (cost_val (v -> VApply.aval) +
                                      cost_val (v -> VApply.apar));
              case TagVList: { int ix;
                               val_list vl = v -> VList.l;
                               int sum = 0;
                               for (ix = 0; ix < vl -> sz; ix++)
                                  sum += cost_val (vl -> arr[ix]);
                               return (sum);
                             };
              case TagVAtom: return (1 + cost_val (v -> VAtom.atcpar));
              default: /* complain */
            };
        };

main ()
        { int cost, the_def;
          alldefs = fscan_def_list (stdin);
          the_def = choose_last_system_def (dl);
          cost = cost_def (the_def);
          printf ("cost of definition %s is %d gates\n",
                        the_def -> DefVal.valnm -> sym, cost);
        };
```

5.7 Final remarks

An overview of the implementation of the describing environment is presented. Various implementation (especially portability) problems and the solutions to these problems chosen by the project team have been discussed.

In particular the generation of necessary datastructure definitions and support routines from one definition file eases the development of semantic functions and transformation tools considerably.

Chapter 6

Discrete timing function models

O Elbereth Gilthoniel
We still remember, we who dwell
in this far land beneath the seas
Thy starlight on the Western Seas
Lord of the Rings J R R Tolkien

6.1 Introduction

Our previous models concerning behaviour only handled static behaviour The static behaviour of a system is often quite simplistic For other systems it may even not be meaningful, for instance systems that contain feedback do not exhibit any static behaviour In this chapter (and later ones) we will take a closer look at semantic functions handling dynamic behaviour and the corresponding models

One abstracted form of dynamic behaviour is of particular interest if one considers the values of the inputs and outputs at fixed equidistant instants in time This model is especially useful for designing synchronous sequential circuits, whose behaviour is governed by an external periodic clock and thus only change state at certain fixed moments in time

We will denote our value domain by $\mathcal{V}$ We will model the domain of interest as discrete time functions, taking $\mathcal{D} = \mathbb{N} \to \mathcal{V}$ with the convention that the value of signal x at time t is $x\ t$ This model will be called the *discrete time function* model

6.2 The generic model revisited

For synchronous sequential circuits it is necessary to introduce some new atomic components, namely the flipflops In the following sections we will assume that the clock input of these flipflops is an implicit input whose 'ticks' indicate the sampling of new instantaneous signal values at the inputs

The introduction of new atomic components is achieved by the following declaration in the Glass text:

Atom
$dff \in \mathbf{Bool} \to E \Rightarrow E,$
$jkff \in \mathbf{Bool} \to E^2 \Rightarrow E,$
$tff \in \mathbf{Bool} \to E \Rightarrow E;$

We will use the extra value parameter of type **Bool** to denote the initial state of the flipflops. This can be seen as analogous to the "value" of a resistor or as an abstract representation of hidden reset circuitry. As we are also going to allow feedback loops in the descriptions, semantic functions must map local definitions in the proper way. Because of these requirements we must extend the generic model in the following way.

Again we start with the transduction of the types. Let E be the base type used in the Glass descriptions and $\mathcal{D}$ be the domain of interpretation:

def $\tau_T : \mathcal{T}_{FDGl} \to \mathcal{T}$
with $\tau_T\ [\![E]\!] = \mathcal{D}$
$\tau_T\ [\![\mathbf{Bool}]\!] = \mathbb{B}$
$\tau_T\ [\![\mathbf{Int}]\!] = \mathbf{Z}$
$\tau_T\ [\![\mathbf{Float}]\!] = \mathbf{R}$
$\tau_T\ [\![T_0\ \&\ \ldots\ \&\ T_{n-1}]\!] = \tau_T\ [\![T_0]\!] \times \ldots \times \tau_T\ [\![T_{n-1}]\!]$
$\tau_T\ [\![U \Rightarrow V]\!] = \tau_T\ [\![U]\!] \to \tau_T\ [\![V]\!]$
$\tau_T\ [\![U \to V]\!] = \tau_T\ [\![U]\!] \to \tau_T\ [\![V]\!]$

Let $\mathcal{A}$ be the set of atoms, and τ_A be the interpretation of the atoms. Then:

$$\forall(a : \mathcal{A} \bullet \tau_A\ [\![a]\!] \in \tau_T\ [\![\mathit{typeof}\ a]\!])$$

For example:

$$\tau_A\ [\![jkff]\!] \in \mathbf{B} \to \mathcal{D}^2 \to \mathcal{D}$$

We now express a semantic function by the following definitions:

def $\tau_E : \mathcal{E}_{Gl} \to \mathcal{I} \to \{\tau_T\}$
with $\tau_E\ [\![v]\!]\ i = i\ v$ if v is a variable
$\tau_E\ [\![e\ \mathbf{where}\ l_0 = e_0; \ldots; l_{n-1} = e_{n-1}\ \mathbf{endwhere}]\!]\ i =$
$\tau_E\ [\![e]\!]\ i'\ \mathbf{where}$
$l_0 = \tau_E\ [\![e_0]\!]\ i'$
$\vdots$
$l_{n-1} = \tau_E\ [\![e_{n-1}]\!]\ i'$
$i'\ v = (v = l_0)?l_0 \dagger (v = l_1)?l_1 \dagger \ldots \dagger (v = l_{n-1})?l_{n-1} \dagger i\ v$
$\tau_E\ [\![a\ c_0\ c_1\ \ldots\ c_{m-1}\ e]\!]\ i = \tau_A[\![a]\!]\ c_0\ c_1\ \ldots\ c_{n-1}\ (\tau_E\ [\![e]\!]\ i)$
if $a \in A$ and all c_i are constants in the value domains
$\tau_E\ [\![f\ e]\!]\ i = \tau_F\ [\![f]\!]\ i\ (\tau_E\ [\![e]\!]\ i)$
$\tau_E\ [\![(e_0, e_1, \ldots, e_{n-1})]\!]\ i = (\tau_E\ [\![e_0]\!]\ i, \tau_E\ [\![e_1]\!]\ i, \ldots, \tau_E\ [\![e_{n-1}]\!]\ i)$

Remark that we have used the fact that local definitions in Funmath are bound to their surrounding definitions. Remember that we only allow constant parametrization of atoms in the kernel language. As one can see these constant parameters become actual arguments of the predefined semantics of the atoms.

$$\begin{array}{l}\textbf{def}\ \ \tau_F : \mathcal{F}_{GI} \to \mathcal{I} \to \{\tau_T\} \\ \textbf{with}\ \tau_F\ [\![\lambda\ t.e]\!]\ i : \tau_T[\![typeof\ (\lambda\ t.e)]\!] \qquad\qquad \text{with} \\ \qquad \tau_F\ [\![\lambda\ t.e]\!]\ i\ d = \tau_E\ [\![e]\!]\ i[\,{}^{d}/_{t}] \\ \qquad \tau_F\ [\![\lambda\ \langle t_0, t_1, \ldots, t_{n-1}\rangle.e]\!]\ i : \tau_T[\![typeof\ (\lambda\ \langle t_0, t_1, \ldots, t_{n-1}\rangle.e)]\!] \qquad \text{with} \\ \qquad \tau_F\ [\![\lambda\ \langle t_0, t_1, \ldots, t_{n-1}\rangle.e]\!]\ i\ (d_0, d_1, \ldots, d_{n-1}) = \\ \qquad\quad \tau_E\ [\![e]\!]\ i[\,{}^{d_0}/_{t_0}] \ldots [\,{}^{d_{n-1}}/_{t_{n-1}}]\end{array}$$

where:

$$\begin{array}{l}\textbf{def}\ \ -[\,{}^{d}/_{t}] : \mathcal{I} \to \mathcal{I} \\ \textbf{with}\ i\,[\,{}^{d}/_{t}]\ t' = \ (t' = t)?d{+}i\ t'\end{array}$$

The meaning of a system then is:

$$\begin{array}{ll}\textbf{def}\ \ \tau_S : \mathcal{F}_{GI} \to \{\tau_T\} & \\ \textbf{with}\ \tau_S\ [\![s]\!] = \tau_F\ [\![s]\!]\ i & \text{where } i \text{ may be any interpretation}\end{array}$$

6.3 Direct extension

The properties of the gates in the discrete time function model are best described using *direct extension*. Direct extension promotes functions into functionals in the following way. By direct extension towards a set X we mean application of the following four definitions:

$$\textbf{poly}\ A, B : \mathcal{T}^2\textbf{.}\ \ \textbf{def}\ \ \overline{-} : (A \to B) \to (X \to A) \to (X \to B)\quad \textbf{with}\ \overline{f}\ s\ t = f\ (s\ t)$$

$$\textbf{poly}\ A, B, k : \mathcal{T} \times \mathcal{T} \times \mathbf{N}\textbf{.}\ \ \textbf{def}\ \ \overset{>}{-} : (A^k \to B) \to (X \to A)^k \to (X \to B)\quad \textbf{with}\ \overset{>}{f}\ s\ t = f\ (T\ s\ t)$$

$$\textbf{poly}\ A, B, l : \mathcal{T} \times \mathcal{T} \times \mathbf{N}\textbf{.}\ \ \textbf{def}\ \ \overset{<}{-} : (A \to B^l) \to (X \to A) \to (X \to B)^l\quad \textbf{with}\ \overset{<}{f}\ s\ i\ t = f\ (s\ t)\ i$$

$$\textbf{poly}\ A, B, k, l : \mathcal{T} \times \mathcal{T} \times \mathbf{N} \times \mathbf{N}\textbf{.}\ \ \textbf{def}\ \ \overset{=}{-} : (A^k \to B^l) \to (X \to A)^k \to (X \to B)^l\quad \textbf{with}\ \overset{=}{f}\ s\ i\ t = f\ (T\ s\ t)\ i$$

where the transposition operator T is defined as follows:

$$\textbf{poly}\ A,B,C : \mathcal{T}^3 \bullet \textbf{def}\ \ T : (A \to B \to C) \to (B \to A \to C)$$
$$\textbf{with}\ T\ f\ b\ a = f\ a\ b$$

Remark that $T^2 = id_{A \to B \to C}$.

Proposition 6.1 *For f and s of appropriate type, the following equalities hold:*

$$\overline{f}\ s = f \circ s$$
$$\overset{>}{f}\ s = f \circ T\ s$$
$$\overset{<}{f}\ s = T\ (f \circ s)$$
$$\overline{\overline{f}}\ s = T\ (f \circ T\ s)$$

We will only prove the last one. The others may be proven likewise.

Proof
Let $f \in A^k \to B^l$ and $s \in (X \to A)^k$. Then:

$$\begin{array}{ll} & \overline{\overline{f}}\ s\ i\ t \\ = \{\text{def } \overline{=}\} & f\ (T\ s\ t)\ i \\ = \{\text{def } \circ\} & (f \circ T\ s)\ t\ i \\ = \{\text{def } T\} & T\ (f \circ T\ s)\ i\ t \end{array}$$

■

6.4 The properties of the atoms

In the discrete time function model we will use $\mathbb{B}$ as value domain (although one can quite easily adapt the model to other value domains like $\mathbb{E}$). Our domain of interest is therefore $\mathcal{D} = \mathbb{N} \to \mathbb{B}$. First we define the properties of the gates by direct extension of the properties of the gates in the static model towards $\mathbb{N}$:

$$\textbf{def}\ \ dtm\_not : (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B} \qquad \tau_A\ [\![not]\!] = dtm\_not$$
$$\textbf{with}\ dtm\_not = \overline{s\_not}$$

$$\textbf{def}\ \ dtm\_and : (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B} \qquad \tau_A\ [\![and]\!] = dtm\_and$$
$$\textbf{with}\ dtm\_and = \overset{>}{s\_and}$$

$$\textbf{def}\ \ dtm\_xor : (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B} \qquad \tau_A\ [\![xor]\!] = dtm\_xor$$
$$\textbf{with}\ dtm\_xor = \overset{>}{s\_xor}$$

Gates therefore have no delay in this model.

Next we define the properties of the flipflops starting with the D-flipflop:

$$\begin{array}{l}\textbf{def}\quad dtm\_dff : \mathbb{B} \to (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B} \\ \textbf{with}\ dtm\_dff\ q\ d\ 0 = q \\ \qquad dtm\_dff\ q\ d\ n = d\ (n-1)\end{array}$$

or alternatively using an auxiliary function *tail*:

$$\begin{array}{l}\textbf{def}\quad tail : (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B} \\ \textbf{with}\ tail\ d\ n = d\ (n+1)\end{array}$$

$$\begin{array}{l}\textbf{def}\quad dtm\_dff : \mathbb{B} \to (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B} \\ \textbf{with}\ dtm\_dff\ q\ d\ 0 = q \\ \qquad dtm\_dff\ q\ d\ n = dtm\_dff\ (d\ 0)\ (tail\ d)\ (n-1)\end{array}$$

Although these definitions look different, they define the same function. The second definition expresses more intuitively the effect of the tick of the clock by indicating the next state of the flipflop. The first definition expresses that a D-flipflop actually performs a delay of one tick of the clock. We will prove the equality of these definitions by induction. In this proof we will indicate the definitions of this function by dtm_dff_1 and dtm_dff_2 respectively.

Theorem 6.2

$$\forall(q, d, n : \mathbb{B} \times (\mathbb{N} \to \mathbb{B}) \times \mathbb{N} \bullet dtm\_dff_1\ q\ d\ n = dtm\_dff_2\ q\ d\ n)$$

Proof (Induction on n)

Base step

$$dtm\_dff_1\ q\ d\ 0 = q = dtm\_dff_2\ q\ d\ 0$$

Induction hypothesis

$$dtm\_dff_1\ q\ d\ n = dtm\_dff_2\ q\ d\ n$$

Induction step

$$\begin{array}{ll} & dtm\_dff_2\ q\ d\ (n+1) \\ = & \{\text{def } dtm\_dff_2\}\ dtm\_dff_2\ (d\ 0)\ (tail\ d)\ n \\ = & \{\text{ind. hyp}\}\ dtm\_dff_1\ (d\ 0)\ (tail\ d)\ n \\ = & \{\text{def } dtm\_dff_1\}\ (tail\ d)\ (n-1) \\ = & \{\text{def } tail\}\ d\ n \\ = & \{\text{def } dtm\_dff_1\}\ dtm\_dff1\ q\ d\ (n+1)\end{array}$$

■

The properties of the JK-flipflop may be defined likewise:

$$\begin{array}{l}\textbf{def}\quad dtm\_jkff : \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B} \\ \textbf{with}\ dtm\_jkff\ q\ (j,k)\ 0 = q \\ \qquad dtm\_jkff\ q\ (j,k)\ n = dtm\_jkff\ new\ (tail\ j, tail\ k)\ (n-1) \\ \qquad\quad \textbf{where}\ new = q?\neg\ (k\ 0)\dagger j\ 0\end{array}$$

or equivalently:

$$\begin{array}{l}\textbf{def}\quad dtm\_jkff : \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B} \\ \textbf{with}\ dtm\_jkff\ q\ (j,k)\ 0 = q \\ \qquad dtm\_jkff\ q\ (j,k)\ n = dtm\ jkff\ q\ (j,k)\ (n-1)?\neg\ (k\ (n-1))\dagger j\ (n-1)\end{array}$$

The properties of the T-flipflop are analogous:

$$\begin{array}{l}\textbf{def}\quad dtm\_tff : \mathbb{B} \to (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B} \\ \textbf{with}\ dtm\_tff\ q\ t\ 0 = q \\ \qquad dtm\_tff\ q\ t\ n = dtm\_tff\ new\ (tail\ t)\ (n-1) \\ \qquad\quad \textbf{where}\ new = t?\neg\ q\dagger q\end{array}$$

or

$$\begin{array}{l}\textbf{def}\quad dtm\_tff : \mathbb{B} \to (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B} \\ \textbf{with}\ dtm\_tff\ q\ t\ 0 = q \\ \qquad dtm\_tff\ q\ t\ n = dtm\ tff\ q\ t\ (n-1) \oplus t\ (n-1)\end{array}$$

We will prove the equivalence of the first pair of these definitions. The other may be proven in an analogous way. In the proof we will indicate the two definitions by dtm_jkff_1 and dtm_jkff_2 respectivily.

Theorem 6.3

$$\forall (q,(j,k),n : \mathbb{B} \times (\mathbb{N} \to \mathbb{B})^2 \times \mathbb{N}\,.\, dtm\_jkff_1\ q\ (j,k)\ n = dtm\_jkff_2\ (j,k)\ d\ n)$$

Proof (Induction on n)

Base step

$dtm\_jkff_1\ q\ (j,k)\ 0 = q = dtm\_jkff_2\ q\ (j,k)\ 0$

Also

$$\begin{array}{lll} & & dtm\_jkff_1\ q\ (j,k)\ 1 \\ = & \{\text{def } dtm\_jkff_1\} & dtm\_jkff_1\ new\ (tail\ j, tail\ k)\ 0\ \textbf{where}\ new = q?\neg(k\ 0)\dagger j\ 0 \\ = & \{\text{def } dtm\_jkff_1\} & new\ \textbf{where}\ new = q?\neg(k\ 0)\dagger j\ 0 \\ = & \{\text{subst.}\} & q?\neg(k\ 0)\dagger j\ 0 \\ = & \{\text{def } dtm\_jkff_2\} & dtm\ jkff_2\ q\ (j,k)\ 0?\neg(k\ 0)\dagger j\ 0 \\ = & \{\text{def } dtm\_jkff_2\} & dtm\ jkff_2\ q\ (j,k)\ 1\end{array}$$

Induction hypothesis

$dtm\_jkff_1\ q\ d\ n = dtm\ jkff_2\ q\ d\ n$

Induction step

$$
\begin{array}{ll}
 & dtm\_jkff_1\ q\ (j,k)\ (n+1) \\
= \{\text{def } dtm\_jkff_1\} & dtm\_jkff_1\ new\ (tail\ j, tail\ k)\ n \ \textbf{where}\ new = q?\neg(k\ 0)\dagger j\ 0 \\
= \{\text{Ind.hyp.}\} & dtm\_jkff_2\ new\ (tail\ j, tail\ k)\ n \ \textbf{where}\ new = q?\neg(k\ 0)\dagger j\ 0 \\
= \{\text{def } dtm\_jkff_2\} & dtm\_jkff_2\ new\ (tail\ j, tail\ k)\ (n-1)?\neg(tail\ k\ (n-1))\dagger tail\ j\ (n-1) \\
 & \quad \textbf{where}\ new = q?\neg(k\ 0)\dagger j\ 0 \\
= \{\text{def } tail\} & dtm\_jkff_2\ new\ (tail\ j, tail\ k)\ (n-1)?\neg(k\ n)\dagger j\ n \\
 & \quad \textbf{where}\ new = q?\neg(k\ 0)\dagger j\ 0 \\
= \{\text{Ind.hyp.}\} & dtm\_jkff_1\ new\ (tail\ j, tail\ k)\ (n-1)?\neg(k\ n)\dagger j\ n \\
 & \quad \textbf{where}\ new = q?\neg(k\ 0)\dagger j\ 0 \\
= \{\text{def } dtm\_jkff_1\} & dtm\_jkff_1\ q\ (j,k)\ n?\neg(k\ n)\dagger j\ n \\
= \{\text{Ind.hyp.}\} & dtm\_jkff_2\ q\ (j,k)\ n?\neg(k\ n)\dagger j\ n \\
= \{\text{def } dtm\_jkff_2\} & dtm\_jkff_2\ q\ (j,k)\ (n+1)
\end{array}
$$

■

We define:

$$
\begin{array}{l}
\tau_A\ [\![dff]\!] = dtm\_dff \\
\tau_A\ [\![jkff]\!] = dtm\ jkff \\
\tau_A\ [\![tff]\!] = dtm\_tff
\end{array}
$$

Again we can use the conformity to our generic model to derive the corresponding semantic function, which we will denote by *dtm*.

Consider for instance the following Glass description of a parity checker:

$$
\begin{array}{l}
\textbf{def}\ \ parity \in E \Rightarrow E; \\
\qquad parity\ x = y\ \textbf{where}\ y = dff\ 0\ (xor\ \langle x, y\rangle);\ \textbf{endwhere}
\end{array}
$$

By applying *dtm* to this description one obtains:

$$
\begin{array}{l}
(dtm\ [\![parity]\!]) : (\mathbf{N} \to \mathbf{B}) \to \mathbf{N} \to \mathbf{B} \\
(dtm\ [\![parity]\!])\ x = y\ \textbf{where}\ y = dtm\ dff\ 0\ (dtm\ xor\ (x, y))
\end{array}
$$

6.5 Explicit clock signals

In our previous model, the flipflops were clocked implicitly: at every step in time they received a tick of the clock. However, for certain classes of circuits this is not a valid assumption, e.g. when flipflops are clocked by (other) subparts of the system. Consider for instance a counter built as a chain of $JK-$flipflops in which every flipflop is clocked by the output of its predecessor in the chain.

Moreover in many synchronous MOSFET networks one often encounters only gates and dynamic latches. Such a dynamic storage element consists of a parasitic capacitor which may be charged or discharged by a pass transistor in NMOS technology or by a transmission gate in CMOS technology. In most cases it is followed by a (signal

restoring) invertor. It may thus be modelled in this model by a transparent latch followed by an invertor.

For these reasons we will extend our previous model by allowing explicit clock signals. We can incorporate explicit clock signals in the discrete time function model by sampling the instantaneous values of the signals at a (sufficiently) high rate and by changing the properties of the flipflops accordingly. As the flipflops will have an extra clock input, their Glass declaration also changes:

$$\begin{aligned}\textbf{atom}\quad & \mathit{dff} \in \mathbf{Bool} \to E^2 \Rightarrow E,\\ & \mathit{jkff} \in \mathbf{Bool} \to E^3 \Rightarrow E,\\ & \mathit{tff} \in \mathbf{Bool} \to E^2 \Rightarrow E,\\ & \mathit{tlatch} \in \mathbf{Bool} \to E^2 \Rightarrow E;\end{aligned}$$

We must also redefine the properties of our atoms. Some flipflops are level sensitive whereas others are positive edge or negative edge triggered. In the following definitions we will encounter examples of all three types. We define the properties of a positive edge triggered D-flipflop:

$$\begin{aligned}&\textbf{def}\quad \mathit{dtm\_dff} : \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B}\\ &\textbf{with}\quad \mathit{dtm\_dff}\ q\ (ck, d)\ 0 = q\\ &\qquad \mathit{dtm\_dff}\ q\ (ck, d)\ n = (ck\ n \wedge \neg(ck\ (n-1)))?d\ (n-1)\dagger \mathit{dtm\_dff}\ q\ (ck, d)\ (n-1)\end{aligned}$$

The following examples of a JK-flipflop and a T-flipflop are clocked by the downgoing edge of the clock:

$$\begin{aligned}&\textbf{def}\quad \mathit{dtm\_jkff} : \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^3 \to \mathbb{N} \to \mathbb{B}\\ &\textbf{with}\quad \mathit{dtm\_jkff}\ q\ (ck, j, k)\ 0 = q\\ &\qquad \mathit{dtm\ jkff}\ q\ (ck, j, k)\ n = (ck\ (n-1) \wedge \neg(ck\ n))?(q'?\neg\ (k\ (n-1))\dagger j\ (n-1))\dagger q'\\ &\qquad\qquad \textbf{where}\ q' = \mathit{dtm\_jkff}\ q\ (ck, j, k)\ (n-1)\end{aligned}$$

$$\begin{aligned}&\textbf{def}\quad \mathit{dtm\_tff} : \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^2 \to \mathbf{N} \to \mathbb{B}\\ &\textbf{with}\quad \mathit{dtm\_tff}\ q\ (ck, t)\ 0 = q\\ &\qquad \mathit{dtm\_tff}\ q\ (ck, t)\ n = (ck\ (n-1) \wedge \neg(ck\ n))?q' \oplus (t\ (n-1))\dagger q'\\ &\qquad\qquad \textbf{where}\ q' = \mathit{dtm\_tff}\ q\ (ck, t)\ (n-1)\end{aligned}$$

The following definition defines the properties of a transparant latch, that is a storage element whose clock input is level sensitive.

$$\begin{aligned}&\textbf{def}\quad \mathit{dtm\_tlatch} : \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^2 \to \mathbf{N} \to \mathbb{B}\\ &\textbf{with}\quad \mathit{dtm\_tlatch}\ q\ (s, i)\ n = s\ n?i\ n\dagger(n = 0)?q\dagger \mathit{dtm\_tlatch}\ q\ (s, i)\ (n-1)\end{aligned}$$

6.6 First implementations

Initial experiments on mapping towards a programming language were based upon the functional programming language Miranda [Tur85]. In the first experiment the sequence types ($\mathbf{N} \to \mathbb{B}$) were mapped on Miranda lists (`[bool]`). Gates were defined correspondingly:

```
seq_xor :: ([bool],[bool]) -> [bool]
seq_xor ([],k) = []
seq_xor (k,[]) = []
seq_xor (a:f,b:g) = s_xor(a,b):seq_xor(f,g)
```

The D-flipflop properties may be defined as:

```
seq_dff :: bool -> [bool] -> [bool]
seq_dff q l = q:l
```

or as:

```
seq_dff2 :: bool -> [bool] -> [bool]
seq_dff2 q [] = [q]
seq_dff2 q (a:l) = q:seq_dff2 a l
```

Thus:

```
seq_parity :: [bool] -> [bool]
seq_parity x = y where y = seq_dff False (seq_xor (x,y))
```

It was found, however, that if one uses the second definition of the *dff* the implementation fails, because the Miranda interpreter needs to know that y is nonempty in order to correctly do the pattern matching to select the second alternative of the definition of *seq dff2* which will then evaluate the first element of y. In Miranda terminology: these definitions constitute a so called "Black Hole". Likewise problems occur when defining the other types of flipflops.

For this reason we did not pursue implementing sequence types as lists in Miranda and turned towards implementing them as function types, because it is fully equivalent from the behavioural point of view and does not cause the problems mentioned above.

Implementing the definitions of the previous sections as higher order functions was even easier than mapping them on functions operating on lists. Because of the lazy evaluation mechanism in the language implementation the problems of the list implementation did not occur. The only remaining disadvantage is that this mechanism in the existing Miranda implementation is slow. Experiments showed that for $n = 30$ one had to wait several minutes before $(dtm\ [\![parity]\!])\ i\ n$ was evaluated for any input i.

6.7 An efficient implementation

As we have seen, the discrete time function model is entirely adequate for synchronous circuits. The only problem that must be solved before such a semantic function can be used in practice is the slow evaluation of our discrete time functions in a functional language (Miranda), due to the necessity of a lazy evaluation.

We will try and find a solution to this problem by reconsidering the image of the parity checker definition under the semantic function *dtm*:

$$(dtm\ [\![parity]\!]) : (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$(dtm\ [\![parity]\!])\ x = y \textbf{ where } y = dtm\_dff\ 0\ (dtm\_xor\ (x, y))$$

A first and second transformation consists of lambda lifting the where clause and introducing the time parameter n by η-conversion:

$$(dtm\ [\![parity]\!]) : (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$(dtm\ [\![parity]\!])\ x\ t = \_y\ x\ t$$

$$\_y : (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$\_y\ x\ n = dtm\_dff\ 0\ (dtm\_xor\ (x, \_y\ x))\ n$$

As a third step we evaluate the body of $\_y$ partially by using the properties of our atoms:

$$\begin{array}{ll} & \_y\ x\ n \\ = \{\text{def } \_y\} & dtm\_dff\ 0\ (dtm\_xor\ (x, \_y\ x))\ n \\ = \{\text{def } dtm\_dff\} & (n = 0)?0\dagger(dtm\_xor\ (x, \_y\ x)\ (n-1)) \\ = \{\text{def } dtm\_xor\} & (n = 0)?0\dagger(s\_\overset{>}{x}or\ (x, \_y\ x)\ (n-1)) \\ = \{\text{def } \overset{>}{\ }\} & (n = 0)?0\dagger(s\_xor\ (x\ (n-1), \_y\ x\ (n-1)) \end{array}$$

After substituting this form for the body of $\_y$ we obtain the following two definitions:

$$(dtm\ [\![parity]\!]) : (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$(dtm\ [\![parity]\!])\ x\ n = \_y\ x\ n$$

$$\_y : (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$\_y\ x\ n = (n = 0)?0\dagger(s\_xor\ (x\ (n-1), \_y\ x\ (n-1))$$

If we want to use these definitions in a simulation run we must provide the input signal x, which is passed on recursively until it is used. However, if we lift the x to global scope, our definitions are no longer higher order functions:

$$x : \mathbb{N} \to \mathbb{B}$$

$$(dtm\ [\![parity]\!]) : \mathbb{N} \to \mathbb{B}$$
$$(dtm\ [\![parity]\!])\ n = \_y\ n$$

$$\_y : \mathbb{N} \to \mathbb{B}$$
$$\_y\ n = (n = 0)?0\dagger(s\_xor\ (x\ (n-1), \_y\ (n-1))$$

We may now evaluate these definitions using a strict evaluation mechanism: we have removed the necessity of lazy evaluation by partially evaluating the original right-hand side using the properties of the atoms.

6.8 The actual implementation

It is clear that right hand sides occurring in the descriptions can only be partially evaluated in the above fashion if they only contain applications of atoms. To guarantee this, the actual implementation of this semantic function removes all applications of λ-abstractions and user-defined system definitions by means of β-reduction. In effect, this removes all hierarchy in the description. We may view this removal as a kind of flattening. Only one definition containing applications of atoms and where clauses remains. This part was implemented as a separate transformation tool, called *uflat* (unidirectional flattener). Other semantic functions also use this tool as a prepass.

Next all where clauses are lambda lifted followed by the partial evaluation of the bodies of the resulting set of definitions. For each of these definitions a C function is generated. For each of the inputs and outputs of the system a C function is generated that interfaces with a user via the X11 window system. This generation process may be steered by the user by setting flags on the command line. For instance the user may choose whether she wants a discrete time model based on a binary ($\mathbb{B}$) or a ternary ($\mathbb{E}$) value domain. She may also choose between implicit and explicit clocking of the flipflops.

Finally the resulting C program is compiled and executed. Using this program the user can edit the input signals and observe the resulting output signals. For instance a typical run of the program generated by this semantic function for the counter example described on page 62 is shown in figure 6.1

6.9 Final remarks

In this chapter several discrete time function models have been presented as well as a revision of the generic model for semantic functions concerning the directional subset of Glass. Both models with or without an explicit clock were discussed. A functional implementation and a more efficient implementation in C have been presented. The latter one has been developed and implemented in the summer of 1990.

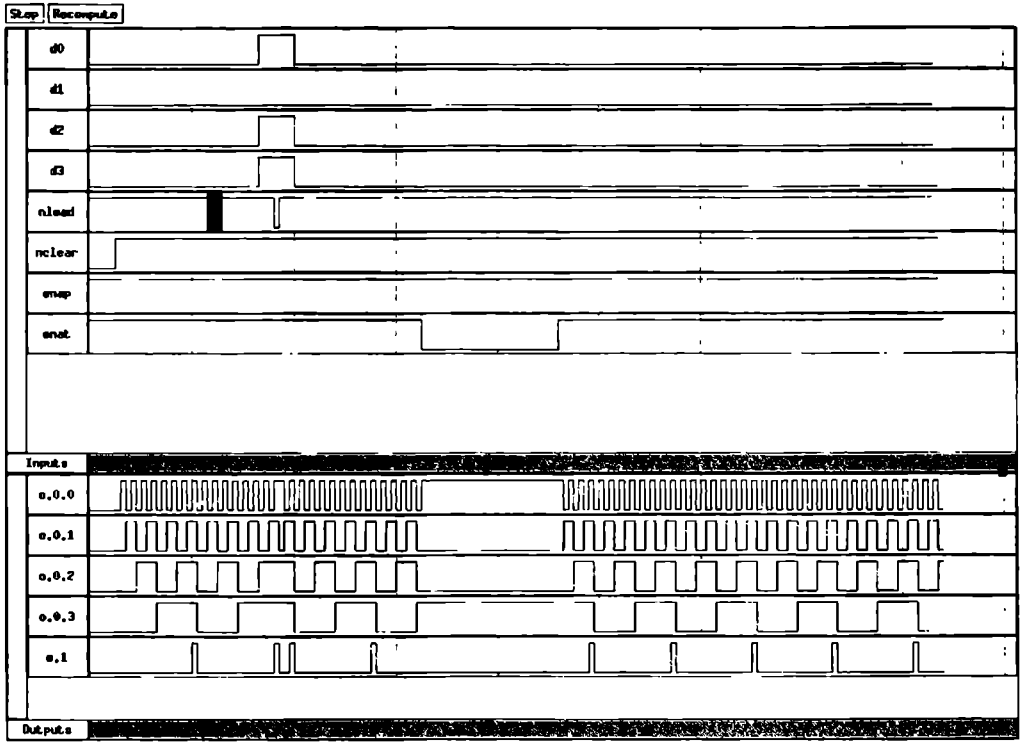


Figure 6.1: Xcounter screendump

Chapter 7

Verification of hardware

The touch of the Sword carried with it
a truth that could not be denied by all
the illusion and deceit of the Warlock lord.
It was a truth he could not admit,
could not accept, could not abide,
yet a truth against which he had no defense.
The Sword of Shannara *Terry Brooks*

7.1 Introduction

The usage of the semantic functions developed thus far has been exemplified only for simulation of certain aspects of the systems described. However, simulation does not constitute proof that certain properties of that hardware hold. To obtain such proofs, hardware must be verified formally.

In this chapter we will introduce a hardware design approach by which hardware *is designed by proving its properties*. We will start the design of a piece of hardware by formally specifying what its high level abstract behaviour should be. The specification is then transformed into more detailed low level characteristics. This process is continued until a description is obtained that fits the properties of the interconnection of certain atomic components. This amounts to obtaining the inverse image under a semantic function, yielding a (correct) Glass description of the specified circuit.

Initially, we will consider only combinational circuits to enhance our intuition on "design by proof". Later we will extend our proofs to synchronous sequential circuits.

A description of the high level abstract behaviour of a certain circuit often takes the form of a function, mapping from an abstract input space into an abstract output space. For instance, the abstract behaviour of an adder may be given as follows:

def $adder : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$
with $adder\ (a, b) = a + b$

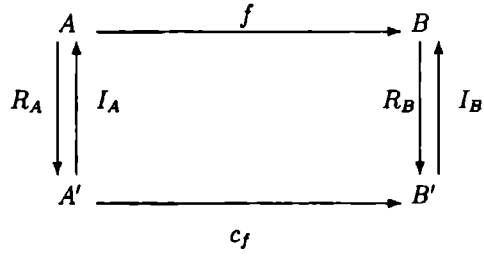
This abstract behaviour is realized by lower level mappings between less abstract do-

mains and codomains. In general high level operations are *realized* by suitable mappings between suitable *representation spaces* to which domain and codomain of the high level operation may be mapped by appropriate *representation functions*.

Consider, for instance, a mapping f from a set A to a set B which may be represented by two concrete domains A' and B' using two representation functions R_A and R_B respectively. To these representation functions correspond two interpretation functions I_A and I_B. We will often require that R_A and I_A be each others' inverse and likewise for R_B and I_B. We will say that a mapping c_f is a *realisation* of f if

$$\forall(a : A . f\ a = I_B\ (c_f\ (R_A\ a)))$$

or, equivalently, if the following diagram commutes:



Since our abstract operations are often operations on natural numbers we will first investigate some of the properties of the (usual) binary representation of the natural numbers. Then we will proceed by proving some combinational circuits and finally do some proofs for sequential circuits.

7.2 Natural number representation

7.2.1 Representation and interpretation functions

We will start by defining the representation and interpretation functions for natural numbers and prove that they are each others' inverse.

We will, however, divert from the usual convention of writing the least significant bit in a word in the rightmost position. This convention in our number system is a consequence of the fact that our current way of writing numbers was adopted in Western Europe in the thirteenth century from the Moors on the Spanish peninsula. Since one writes from right to left in arabic, the least significant digit is written first, which immediately specifies its weight. Because we write from left to right an onlooker can not know what the weight of a digit will be at the moment of writing: it still depends on the number of digits that follows the current one. Strangely enough, the arabic notation for numbers was derived from Indian matematicians which wrote numbers from left to right writing the least significant digit indeed first.

In hardware design, one is actually more concerned with facilitating the formulation of proofs than in the actual order of bits in a word (One can always interchange wires). It therefore seems advisable to adopt the (original) Indian view on numbers.

This also fits nicely with the Funmath convention of viewing lists as functions. Consider the list $a = (a_0, a_1, \ldots, a_{n-1})$ then $a\ i = a_i$ which we may give weight 2^i, whereas we would have to give it weight 2^{n-1-i} if we would place the least significant digit rightmost. This leads to the following definition of the interpretation function:

$$\begin{array}{l}\textbf{def}\quad I_u : \mathbf{N} \ni n \to \mathbf{B}^n \to \square 2^n \\ \textbf{with}\ I_u\ 0\ \varepsilon = 0 \\ \qquad I_u\ n\ a = a\ 0 + 2 \cdot I_u\ (n-1)\ (\sigma\ a)\end{array}$$

Proposition 7.1 $\forall(n, a : \mathbf{N} \times \mathbf{B}^n \cdot I_u\ n\ a = \sum(i : \square n \cdot a\ i \cdot 2^i))$

Proof (Induction on n)

Base step

$$\begin{array}{ll} & I_u\ 0\ \varepsilon \\ = \{\text{def } I_u\} & 0 \\ = \{\text{def } \textstyle\sum\} & \textstyle\sum \varepsilon \\ = \{\text{logic}\} & \textstyle\sum(i : \emptyset \cdot \varepsilon\ i \cdot 2^i) \\ = \{\text{def } \square\} & \textstyle\sum(i : \square 0 \cdot \varepsilon\ i \cdot 2^i)\end{array}$$

Induction hypothesis

$I_u\ n\ a = \sum(i : \square n \cdot a\ i \cdot 2^i)$

Induction step

$$\begin{array}{ll} & I_u\ (n+1)\ a \\ = \{\text{def } I_u\} & a\ 0 + 2 \cdot I_u\ n\ (\sigma\ a) \\ = \{\text{ind. hyp}\} & a\ 0 + 2 \cdot \textstyle\sum(i : \square n \cdot \sigma\ a\ i \cdot 2^i) \\ = \{\text{def } \sigma\} & a\ 0 + 2 \cdot \textstyle\sum(i : \square n \cdot a\ (i+1) \cdot 2^i) \\ = \{\text{arith.}\} & a\ 0 + \textstyle\sum(i : \square n \cdot a\ (i+1) \cdot 2^{i+1}) \\ = \{\text{def } \textstyle\sum\} & \textstyle\sum(i : \square (n+1) \cdot a\ i \cdot 2^i)\end{array}$$

■

The following two properties of the interpretation function can also be proven in a similar way.

$$I_u\ (n+1)\ (a \succ A) = a\ 0 + 2 \cdot I_u\ n\ A$$
$$I_u\ (n+1)\ (A \prec a) = a \cdot 2^n + I_u\ n\ A$$

We define the representation function as follows:

def $R_u : \mathbf{N} \ni n \to \Box 2^n \to \mathbb{B}^n$
with $R_u\ 0\ 0 = \varepsilon$
$R_u\ n\ a = (a \textbf{ mod } 2) \succ R_u\ (n-1)\ (a \textbf{ div } 2)$

Remark that this definition is wellformed since $m \in \Box 2^n \Rightarrow m \textbf{ div } 2 \in \Box 2^{n-1}$.

Proposition 7.2 $\forall(n, a, i : \mathbf{N} \ni n \times \Box 2^n \times \Box n \,.\, R_u\ n\ a\ i = (a \textbf{ div } 2^i) \textbf{ mod } 2)$

Proof (Induction on i)

Basestep

$$
\begin{array}{ll}
 & R_u\ n\ a\ 0 \\
= \{\text{def } R_u\} & ((a \textbf{ mod } 2) \succ R_u\ (n-1)\ (a \textbf{ div } 2))\ 0 \\
= \{\text{prop} \succ\} & a \textbf{ mod } 2 \\
= \{\text{arith}\} & (a \textbf{ div } 2^0) \textbf{ mod } 2
\end{array}
$$

Induction hypothesis
$R_u\ n\ a\ i = (a \textbf{ div } 2^i) \textbf{ mod } 2$

Induction step

$$
\begin{array}{ll}
 & R_u\ n\ a\ (i+1) \\
= \{\text{def } R_u\} & ((a \textbf{ mod } 2) \succ R_u\ (n-1)\ (a \textbf{ div } 2))\ (i+1) \\
= \{\text{prop} \succ\} & R_u\ (n-1)\ (a \textbf{ div } 2)\ i \\
= \{\text{ind.hyp}\} & ((a \textbf{ div } 2) \textbf{ div } 2^i) \textbf{ mod } 2 \\
= \{\text{arith}\} & (a \textbf{ div } 2^{i+1}) \textbf{ mod } 2
\end{array}
$$

■

Theorem 7.3 $I_u\ n = (R_u\ n)^{-}$

Proof (Induction on n)

Base step

$$
\begin{array}{ll}
 & R_u\ 0\ (I_u\ 0\ \varepsilon) \\
= \{\text{def } I_u\} & R_u\ 0\ 0 \\
= \{\text{def } R_u\} & \varepsilon
\end{array}
$$

Induction hypothesis
$R_u\ n\ (I_u\ n\ a) = a$

Induction step

$$
\begin{array}{ll}
 & R_u\ (n+1)\ (I_u\ (n+1)\ a) \\
= \{\text{def } I_u\} & R_u\ (n+1)\ (a\ 0 + 2 \cdot I_u\ n\ (\sigma\ a)) \\
= \{\text{def } R_u\} & ((a\ 0 + 2 \cdot I_u\ n\ (\sigma\ a)) \textbf{ mod } 2) \succ R_u\ n\ ((a\ 0 + 2 \cdot I_u\ n\ (\sigma\ a)) \textbf{ div } 2) \\
= \{\text{prop div, mod}\} & a\ 0 \succ R_u\ n\ (I_u\ n\ (\sigma\ a)) \\
= \{\text{ind.hyp}\} & a\ 0 \succ \sigma\ a \\
= \{\text{prop} \succ, \sigma\} & a
\end{array}
$$

Likewise one can prove: $I_u\ n\ (R_u\ n\ b) = b$

■

7.2.2 Binary addition

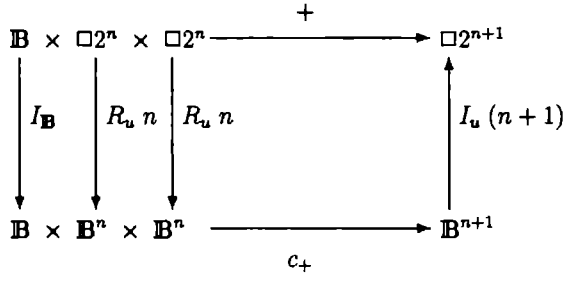
One of the most often used abstract operations in hardware is the addition of two natural numbers. To this operation corresponds the following concrete function:

$$\begin{array}{l}\textbf{def}\quad c_+ : \mathbb{N} \ni n \to \mathbb{B} \times \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^{n+1} \\ \textbf{with}\ c_+\ 0\ (c_{in}, \varepsilon, \varepsilon) = \tau c_{in} \\ \qquad c_+\ n\ (c_{in}, a, b) = ((c_{in} + a\ 0 + b\ 0)\ \textbf{mod}\ 2) \succ \\ \qquad\qquad\qquad\qquad c_+\ (n-1)\ ((c_{in} + a\ 0 + b\ 0)\ \textbf{div}\ 2), \sigma a, \sigma b)\end{array}$$

Theorem 7.4 *c_+ realizes the addition, i.e.*

$$\forall(n, c_{in}, a, b : \mathbb{N} \ni n \times \mathbb{B} \times \square 2^n \times \square 2^n \cdot I_u\ (n+1)\ (c_+\ n\ (c_{in},\ R_u\ n\ a,\ R_u\ n\ b)) = a + b + c_{in})$$

To this theorem corresponds the following diagram:



where $I_{\mathbb{B}}$ is the identity function on $\mathbb{B}$. For proving this theorem, it is convenient to introduce the following lemma.

Lemma 7.5

$$\forall(n, c_{in}, a, b : \mathbb{N} \ni n \times \mathbb{B} \times \mathbb{B}^n \times \mathbb{B}^n \cdot I_u\ (n+1)\ (c_+\ n\ (c_{in}, a, b)) = c_{in} + I_u\ n\ a + I_u\ n\ b)$$

Proof (of lemma by induction on n)

Base step

$$\begin{array}{ll} & I_u\ 1\ (c_+\ 0\ (c_{in}, \varepsilon, \varepsilon)) \\ = \{\text{def } c_+\} & I_u\ 1\ (\tau\ c_{in}) \\ = \{\text{def } I_u\} & \tau\ c_{in}\ 0 + 2.I_u\ 0\ (\sigma\ (\tau\ c_{in})) \\ = \{\text{def } \tau\} & c_{in} + 2.I_u\ 0\ \varepsilon \\ = \{\text{def } I_u\} & c_{in} + I_u\ 0\ \varepsilon + I_u\ 0\ \varepsilon \end{array}$$

Induction hypothesis

$$I_u\ (n+1)\ (c_+\ n\ (c_{in}, a, b)) = c_{in} + I_u\ n\ a + I_u\ n\ b$$

Induction step

$$\begin{array}{lll}
 & & I_u\ (n+2)\ (c_+\ (n+1)\ (c_{in}, a, b)) \\
= & \{\text{def } c_+\} & I_u\ (n+2)\ (((c_{in} + a\ 0 + b\ 0)\ \mathbf{mod}\ 2) \succ \\
 & & \quad c_+\ n\ ((c_{in} + a\ 0 + b\ 0)\ \mathbf{div}\ 2, \sigma a, \sigma b)) \\
= & \{\text{prop } I_u\} & ((c_{in} + a\ 0 + b\ 0)\ \mathbf{mod}\ 2) + \\
 & & \quad 2 \cdot I_u\ (n+1)\ (c_+\ n\ ((c_{in} + a\ 0 + b\ 0)\ \mathbf{div}\ 2, \sigma a, \sigma b)) \\
= & \{\text{ind.hyp}\} & ((c_{in} + a\ 0 + b\ 0)\ \mathbf{mod}\ 2) + \\
 & & \quad 2 \cdot ((c_{in} + a\ 0 + b\ 0)\ \mathbf{div}\ 2 + I_u\ n\ (\sigma a) + I_u\ n\ (\sigma b)) \\
= & \{\text{dist } \cdot\} & ((c_{in} + a\ 0 + b\ 0)\ \mathbf{mod}\ 2) + 2 \cdot ((c_{in} + a\ 0 + b\ 0)\ \mathbf{div}\ 2) + \\
 & & \quad 2 \cdot I_u\ n\ (\sigma a) + 2 \cdot I_u\ n\ (\sigma b) \\
= & \{\text{Euclid}\} & c_{in} + a\ 0 + b\ 0 + 2 \cdot I_u\ n\ (\sigma a) + 2 \cdot I_u\ n\ (\sigma b) \\
= & \{\text{def } I_u\} & c_{in} + I_u\ (n+1)\ a + I_u\ (n+1)\ b
\end{array}$$

■

Proof (of theorem 7.4)

$$\begin{array}{lll}
 & & I_u\ (n+1)\ (c_+\ n\ (c_{in}, R_u\ n\ a, R_u\ n\ b)) \\
= & \{\text{Lemma}\} & c_{in} + I_u\ n\ (R_u\ n\ a) + I_u\ n\ (R_u\ n\ b) \\
= & \{\text{theorem 7.3}\} & c_{in} + a + b
\end{array}$$

■

This theorem can also be interpreted as follows: if we have a component, say *adc*, with static behaviour:

$$\begin{array}{l}
(\mathit{simplex}\ [\![\mathit{adc}]\!]) : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B} \\
(\mathit{simplex}\ [\![\mathit{adc}]\!])\ (c_i,\ a,\ b) = (c_o,\ s) \\
\quad \mathbf{where} \\
\quad\quad c_o = (a + b + c_i)\ \mathbf{div}\ 2 \\
\quad\quad s = (a + b + c_i)\ \mathbf{mod}\ 2
\end{array}$$

then a chain of *adc*'s will add n-bit numbers thus proving the correctness of the following Glass macro:

$$\begin{array}{l}
\mathbf{mac}\ \mathit{adder} \in \mathbf{Int} \ni n \to E^n \times E^n \times E \Rightarrow E \times E^n; \\
\quad \mathit{adder}\ 0\ \langle\langle\rangle,\ \langle\rangle,\ c\rangle = \langle c,\ \langle\rangle\rangle; \\
\quad \mathit{adder}\ n\ \langle a : A,\ b : B,\ c_i\rangle = \langle c_o,\ s : S\rangle \\
\quad\quad \mathbf{where} \\
\quad\quad\quad \langle v, s\rangle = \mathit{adc}\ \langle c_i,\ a,\ b\rangle; \\
\quad\quad\quad \langle c_o, S\rangle = \mathit{adder}\ (n-1)\ \langle A,\ B,\ v\rangle; \\
\quad\quad \mathbf{endwhere}
\end{array}$$

7.2.3 Truncation

We define the following function:

def $Trunc : \mathbb{N} \ni n \rightarrow \mathbb{B}^{n+1} \rightarrow \mathbb{B}^n$
with $Trunc\ n\ (x \prec a) = x$

Remark that:

$$\forall(n, x : \mathbb{N} \ni n \times \mathbb{B}^{n+1} \bullet x = Trunc\ n\ x \prec x\ n).$$

Theorem 7.6 *(Trunc n) realizes* (**mod** 2^n) *restricted to* $\square 2^{n+1}$, *that is*

$$\forall(n \in \mathbb{N}, a \in \square 2^{n+1} \bullet I_u\ n\ (Trunc\ n\ (R_u\ (n+1)\ a)) = a \bmod 2^n)$$

Proof
Let $n \in \mathbb{N}$, $a \in \square 2^{n+1}$. Then:

$$\begin{array}{ll} & a \bmod 2^n \\ = & \{\text{theorem 7.3}\} \quad (I_u\ (n+1)\ (R_u\ (n+1)\ a)) \bmod 2^n \\ = & \{\text{remark}\} \quad (I_u\ (n+1)\ (Trunc\ n\ (R_u\ (n+1)\ a) \prec R_u\ (n+1)\ a\ n)) \bmod 2^n \\ = & \{\text{prop } I_u\} \quad (2^n \cdot (R_u\ (n+1)\ a\ n) + I_u\ n\ (Trunc\ n\ (R_u\ (n+1)\ a))) \bmod 2^n \\ = & \{I_u\ n\ x < 2^n\} \quad I_u\ n\ (Trunc\ n\ (R_u\ (n+1)\ a)) \end{array}$$

■

7.2.4 Special case: derivation of an incrementer

Incrementing a number means adding one to it. An incrementer with an enable input e can therefore be seen as an adder with $c_{in} = e$ and $b = 0$. This leads to the following two definitions:

def $inc : \mathbb{N} \ni n \rightarrow \mathbb{B} \times \square 2^n \rightarrow \square 2^{n+1}$
with $inc\ n\ (e, a) = a + e$

def $c_{inc} : \mathbb{N} \ni n \rightarrow \mathbb{B} \times \mathbb{B}^n \rightarrow \mathbb{B}^{n+1}$
with $c_{inc}\ n\ (e,\ a) = c_+\ n\ (e,\ a,\ R_u\ n\ 0)$

From theorem 7.4 we can immediately conclude the following corollary:

Corollary 7.7 c_{inc} *realizes the enabled increment, that is:*

$$\forall(n, e, a : \mathbb{N} \ni n \times \mathbb{B} \times \square 2^n \bullet I_u\ (n+1)\ (c_{inc}\ n\ (e,\ R_u\ n\ a)) = a + e = inc\ (e, a))$$

Substituting the body of c_+ in the body of c_{inc} yields:

def $c_{inc} : \mathbb{N} \ni n \rightarrow \mathbb{B} \times \mathbb{B}^n \rightarrow \mathbb{B}^{n+1}$
with $c_{inc}\ 0\ (e, \varepsilon) = \tau e$
$\quad c_{inc}\ n\ (e, x) = (e + x\ 0) \bmod 2 \succ c_{inc}\ (n-1)\ ((e + x\ 0)\ \mathbf{div}\ 2, \sigma x)$

Since

$$(e + f)\ \mathbf{mod2} = e \oplus f$$
$$(e + f)\ \mathbf{div2} = e \wedge f$$

we can replace the arithmetic operations in the right hand sides of the equalities in the body of c_{inc} by operations that are directly realizable by gates:

$$\begin{array}{l}\mathbf{def}\quad c_{inc} : \mathbb{N} \ni n \to \mathbb{B} \times \mathbb{B}^n \to \mathbb{B}^{n+1} \\ \mathbf{with}\ c_{inc}\ 0\ (e, \varepsilon) = \tau e \\ \qquad c_{inc}\ n\ (e,\ x) = e \oplus x\ 0 \rightarrowtail c_{inc}\ (n-1)\ (e \wedge x\ 0, \sigma x)\end{array}$$

We can also replace this recursive definition of c_{inc} by a closed-form expression using the following (twofold) proposition:

Proposition 7.8

$$\forall(n, e, x, i : \mathbb{N} \ni n \times \mathbb{B} \times \mathbb{B}^n \times \Box n \,.\, c_{inc}\ n\ (e,\ x)\ i = x\ i \oplus (e \wedge \forall(j : \Box i \,.\, x\ j)))$$
$$\forall(n, e, x : \mathbb{N} \ni n \times \mathbb{B} \times \mathbb{B}^n \,.\, c_{inc}\ n\ (e,\ x)\ n = e \wedge \forall(j : \Box n \,.\, x\ j))$$

Proof (by induction on i)

Base step

$$\begin{array}{lll} & c_{inc}\ n\ (e,\ x)\ 0 \\ = & \{\text{def } c_{inc}\} & (e \oplus x\ 0 \rightarrowtail c_{inc}\ (n-1)\ (e \wedge x\ 0, \sigma x))\ 0 \\ = & \{\text{prop} \rightarrowtail\} & e \oplus x\ 0 \\ = & \{\text{def } \forall\} & x\ 0 \oplus (e \wedge \forall(j : \emptyset \,.\, x\ j)) \\ = & \{\Box 0 = \emptyset\} & x\ 0 \oplus (e \wedge \forall(j : \Box 0 \,.\, x\ j)) \end{array}$$

Induction hypothesis

$$c_{inc}\ n\ (e,\ x)\ i = x\ i \oplus (e \wedge \forall(j : \Box i \,.\, x\ j))$$

Induction step

$$\begin{array}{lll} & c_{inc}\ n\ (e,\ x)\ (i+1) \\ = & \{\text{def } c_{inc}\} & (e \oplus x\ 0 \rightarrowtail c_{inc}\ (n-1)\ (e \wedge x\ 0, \sigma x))\ (i+1) \\ = & \{\text{prop} \rightarrowtail\} & c_{inc}\ (n-1)\ (e \wedge x\ 0, \sigma x)\ i \\ = & \{\text{ind.hyp}\} & (\sigma x\ i) \oplus (e \wedge x\ 0 \wedge \forall(j : \Box i \,.\, \sigma x\ j)) \\ = & \{\text{def } \sigma\} & x\ (i+1) \oplus (e \wedge x\ 0 \wedge \forall(j : \Box i \,.\, x\ (j+1))) \\ = & \{\text{prop } \forall\} & x\ (i+1) \oplus (e \wedge \forall(j : \Box i \,.\, x\ j)) \end{array}$$

The second part of the proposition can be proven in the same way.

■

Again we substitute the closed-form expression of this lemma in the body of our concrete increment function. This gives the following definition:

$$\begin{array}{l}\textbf{def}\quad c_{inc} : \mathbb{N} \ni n \to \mathbb{B} \times \mathbb{B}^n \to \mathbb{B}^{n+1} \\ \textbf{with}\ c_{inc}\ n\ (e,\ x)\ n = e \wedge \forall(j : \Box n \,.\, x\ j) \\ \qquad c_{inc}\ n\ (e,\ x)\ i = x\ i \oplus (e \wedge \forall(j : \Box i \,.\, x\ j))\end{array}$$

We introduce two operations that correspond to compositions of operators introduced previously:

$$\begin{array}{l}\textbf{def}\quad tinc : \mathbb{N} \ni n \to \mathbf{B} \times \Box 2^n \to \Box 2^n \\ \textbf{with}\ tinc\ n\ (e, a) = (a + e)\ \textbf{mod}\ 2^n\end{array}$$

$$\begin{array}{l}\textbf{def}\quad c_{tinc} : \mathbb{N} \ni n \to \mathbb{B} \times \mathbb{B}^n \to \mathbb{B}^n \\ \textbf{with}\ c_{tinc}\ n\ (e,\ x)\ i = x\ i \oplus (e \wedge \forall(j : \Box i \,.\, x\ j))\end{array}$$

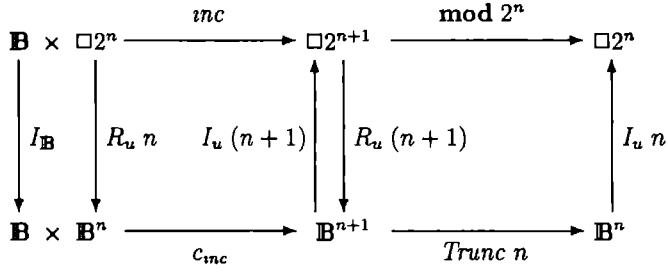
Remark that

$$\begin{array}{l}tinc\ n = (\textbf{mod}\ 2^n) \circ (inc\ n) \\ c_{tinc}\ n = (Trunc\ n) \circ (c_{inc}\ n)\end{array}$$

Theorem 7.9 *$(c_{tinc}\ n)$ realizes the enabled increment modulo 2^n, i.e.*

$$\forall(n, e, a : \mathbb{N} \ni n \times \mathbb{B} \times \Box 2^n \,.\, I_u\ n\ (c_{tinc}\ n\ (e,\ (R_u\ n\ a))) = (a + e)\ \textbf{mod}\ 2^n)$$

Proof
Consider the following diagram:



■

7.3 Towards sequential circuits

7.3.1 The time model

Until now we have only discussed realisation schemes in which the concrete operations correspond to the statical behaviour of certain combinational components. This is no longer possible if we want to prove certain properties of sequential circuits. In this

section we will consider only synchronous sequential circuits. Thus we may model the properties of Glass atoms using discrete time functions.

However, we must define new interpretation and representation functions since the domains and codomains of these functions are now function spaces. As might be expected we define them by direct extension of the interpretation and representation functions of the previous section:

def $I_{ud} : \mathbb{N} \ni n \to (\mathbb{N} \to \mathbb{B})^n \to (\mathbb{N} \to \square 2^n)$
with $I_{ud}\ n = (I_u\ n)^>$

def $R_{ud} : \mathbb{N} \ni n \to (\mathbb{N} \to \square 2^n) \to (\mathbb{N} \to \mathbb{B})^n$
with $R_{ud}\ n = (R_u\ n)^<$

An immediate consequence of these definitions and proposition 6.1 is the following

Corollary 7.10
$\forall(n, x : \mathbb{N} \ni n \times (\mathbb{N} \to \mathbb{B})^n \,.\, I_{ud}\ n\ x = I_u\ n \circ (T\ x))$
$\forall(n, s : \mathbb{N} \ni n \times (\mathbb{N} \to \square 2^n) \,.\, R_{ud}\ n\ s = T\ (R_u\ n \circ s))$

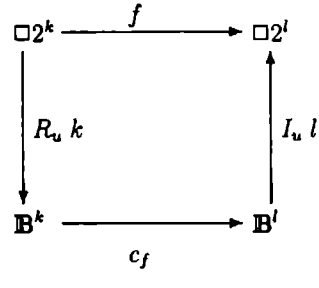
Proposition 7.11 $I_{ud}\ n = (R_{ud}\ n)^-$

Proof
Let $n, s : \mathbb{N} \ni n \times (\mathbb{N} \to \square 2^n)$. Then:

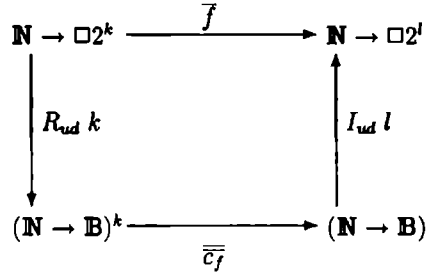
	$I_{ud}\ n\ (R_{ud}\ n\ s)$
$= \{\text{corr. 7.10}\}$	$I_u\ n \circ (T\ (R_{ud}\ n\ s)$
$= \{\text{corr. 7.10}\}$	$I_u\ n \circ (T\ (T\ (R_u\ n \circ s)))$
$= \{T \circ T = id\}$	$I_u\ n \circ R_u\ n \circ s$
$= \{I_u\ n = (R_u\ n)^-\}$	s

That $R_{ud}\ n\ (I_{ud}\ n\ x) = x$ can be proven likewise.
■
A very useful theorem is the following:

Theorem 7.12
If the following diagram commutes



then the direct extension of this diagram also commutes:



Proof

Let $x : \mathbb{N} \to \square 2^k$. Then:

$$
\begin{array}{ll}
 & I_{ud}\ l\ (\overline{\overline{c_f}}\ (R_{ud}\ k\ x)) \\
= \{\text{corr 7.10}\} & I_u\ l \circ (T\ (\overline{c_f}\ (R_{ud}\ k\ x))) \\
= \{\text{prop 6.1}\} & I_u\ l \circ (T\ (T\ (c_f \circ (T\ (R_{ud}\ k\ x))))) \\
= \{\text{corr 7.10}\} & I_u\ l \circ (T\ (T\ (c_f \circ (T\ (T\ (R_u\ k \circ x)))))) \\
= \{T^2 = id\} & I_u\ l \circ c_f \circ R_u\ k \circ x \\
= \{c_f \text{ realizes } f\} & f \circ x \\
= \{\text{prop 6.1}\} & \overline{f}\ x
\end{array}
$$

■

7.3.2 A first step in design

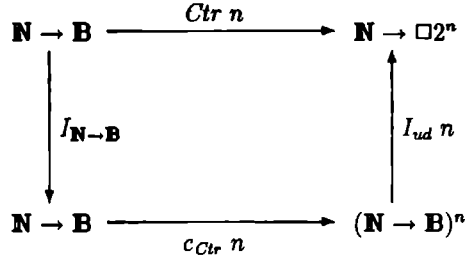
In this section we will design and prove a simple counter circuit. The pure mathematical specification of a counter in the chosen time model is:

$$
\begin{array}{ll}
\textbf{def} & Ctr : (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{N}) \\
\textbf{with} & Ctr\ en\ 0 = 0 \\
 & \quad Ctr\ en\ t = Ctr\ en\ (t-1) + en\ (t-1)
\end{array}
$$

which says that if the input *en* equals 1, the counter should advance and retain its value if its input equals 0. Since hardware is limited to finite realisations we must limit ourselves also in our specification. For this purpose, we will parametrize our definitions. In this way we do not limit ourselves too early in the design process. The specification then becomes:

$$
\begin{array}{ll}
\textbf{def} & Ctr : \mathbb{N} \ni n \to (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \square 2^n) \\
\textbf{with} & Ctr\ n\ en\ 0 = 0 \\
 & \quad Ctr\ n\ en\ t = (Ctr\ n\ en\ (t-1) + en\ (t-1))\ \textbf{mod}\ 2^n
\end{array}
$$

We will designate the corresponding concrete operation realizing Ctr by the name c_{Ctr}. We can derive some of the properties of c_{Ctr} by requiring that the following diagram commutes:



We may therefore define c_{Ctr} by:

def $c_{Ctr} : \mathbb{N} \ni n \to (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})^n$
with $c_{Ctr}\ n\ en = R_{ud}\ n\ (Ctr\ n\ en)$

We will deduce several properties of this function, which will enable us to replace the right hand side of c_{Ctr} by closed-form formulas in which only functions occur that are directly realizable by gates or flipflops.

Lemma 7.13

$$\forall(n, en, t : \mathbb{N} \times (\mathbb{N} \to \mathbb{B}) \times \mathbb{N} \,.\, T\ (c_{Ctr}\ n\ en)\ t = R_u\ n \circ (Ctr\ n\ en\ t))$$

Proof

	$T\ (c_{Ctr}\ n\ en)\ t$
$=$ {def c_{Ctr}}	$T\ (R_{ud}\ n\ (Ctr\ n\ en)\ t$
$=$ {corr 7.10}	$T\ (T\ (R_u\ n \circ (Ctr\ n\ en)))\ t$
$=$ {$T^2 = id$}	$R_u\ n \circ (Ctr\ n\ en)\ t$
$=$ {def $\circ$}	$R_u\ n\ (Ctr\ n\ en\ t)$

The following proposition defines the initialization of the concrete counter.

Proposition 7.14

$$\forall(n, en, i : \mathbb{N} \ni n \times (\mathbb{N} \to \mathbb{B}) \times \square n \,.\, c_{Ctr}\ n\ en\ i\ 0 = 0)$$

Proof

	$c_{Ctr}\ n\ en\ i\ 0$
$=$ {def c_{Ctr}}	$R_{ud}\ n\ (Ctr\ n\ en)\ i\ 0$
$=$ {prop R_{ud}}	$T\ (R_u\ n \circ (Ctr\ n\ en))\ i\ 0$

$$
\begin{array}{ll}
= \{\text{def } T\} & R_u\ n \circ (Ctr\ n\ en)\ 0\ i \\
= \{\text{def } \circ\} & R_u\ n\ (Ctr\ n\ en\ 0)\ i \\
= \{\text{def } Ctr\} & R_u\ n\ 0\ i \\
= \{\text{def } R_u\} & (0\ \mathbf{div}\ 2^i)\ \mathbf{mod}\ 2 \\
= \{\text{arith}\} & 0
\end{array}
$$

■

Lemma 7.15

$$
\begin{array}{l}
\forall(n, en, t : \mathbb{N} \times (\mathbb{N} \to \mathbb{B}) \times \mathbb{N} \mathbin{\&} t > 0 \bullet \\
\quad T\ (c_{Ctr}\ n\ en)\ t = c_{tinc}\ n\ (en\ (t-1),\ T\ (c_{Ctr}\ n\ en)\ (t-1)))
\end{array}
$$

Proof

$$
\begin{array}{ll}
 & T\ (c_{Ctr}\ n\ en)\ t \\
= \{\text{Lemma 7.13}\} & R_u\ n\ (Ctr\ n\ en\ t) \\
= \{\text{def } Ctr\} & R_u\ n\ ((en\ (t-1) + Ctr\ n\ en\ (t-1))\ \mathbf{mod}\ 2^n) \\
= \{\text{theorem 7.9}\} & R_u\ n\ (I_u\ n\ (c_{tinc}\ n\ (en\ (t-1),\ R_u\ n\ (Ctr\ n\ en\ (t-1))))) \\
= \{I_u\ n = (R_u\ n)\ \} & c_{tinc}\ n\ (en\ (t-1),\ R_u\ n\ (Ctr\ n\ en\ (t-1))) \\
= \{\text{Lemma 7.13}\} & c_{tinc}\ n\ (en\ (t-1),\ T\ (c_{Ctr}\ n\ en)\ (t-1))
\end{array}
$$

■

Proposition 7.16

$$
\begin{array}{l}
\forall(n, en, i, t : \mathbb{N} \times (\mathbb{N} \to \mathbf{B}) \times \square n \times \mathbb{N} \mathbin{\&} t > 0 \bullet \\
\quad c_{Ctr}\ n\ en\ i\ t = c_{Ctr}\ n\ en\ i\ (t-1) \oplus (en\ (t-1) \wedge \forall(j : \square i \bullet c_{Ctr}\ n\ en\ j\ (t-1))))
\end{array}
$$

Proof

$$
\begin{array}{ll}
 & c_{Ctr}\ n\ en\ i\ t \\
= \{\text{def } T\} & T\ (c_{Ctr}\ n\ en)\ t\ i \\
= \{\text{lemma 7.15}\} & c_{tinc}\ n\ (en\ (t-1),\ T\ (c_{Ctr}\ n\ en)\ (t-1))\ i \\
= \{\text{def } c_{tinc}\} & T\ (c_{Ctr}\ n\ en)\ (t-1)\ i \oplus (en\ (t-1) \wedge \\
 & \qquad \forall(j : \square i \bullet T\ (c_{Ctr}\ n\ en)\ (t-1)\ j)) \\
= \{\text{def } T\} & c_{Ctr}\ n\ en\ i\ (t-1) \oplus (en\ (t-1) \wedge \forall(j : \square i \bullet c_{Ctr}\ n\ en\ j\ (t-1))
\end{array}
$$

■

By substituting the results of propositions 7.14 and 7.16 in the body of c_{Ctr} we obtain:

$$
\begin{array}{l}
\mathbf{def}\ \ c_{Ctr} : \mathbb{N} \ni n \to (\mathbb{N} \to \mathbf{B}) \to (\mathbb{N} \to \mathbf{B})^n \\
\mathbf{with}\ c_{Ctr}\ n\ en\ i\ 0 = 0 \\
\quad c_{Ctr}\ n\ en\ i\ t = c_{Ctr}\ n\ en\ i\ (t-1)\ \oplus\ (en\ (t-1) \wedge \forall(j : \square i \bullet c_{Ctr}\ n\ en\ j\ (t-1)))
\end{array}
$$

7.3.3 Finding an appropriate subsystem

If we look at the last definition it is clear that we need a component, say *toggle* whose behaviour is defined as follows:

$$\begin{array}{ll}\textbf{def} & toggle : (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B}) \\ \textbf{with} & toggle\ in\ 0 = 0 \\ & \quad toggle\ in\ t = toggle\ in\ (t-1)\ \oplus\ in\ (t-1)\end{array}$$

If we compare this to the discrete time behaviour of the *tff*:

$$\begin{array}{l} dtm\ [\![tff]\!] : \mathbb{B} \to (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B} \\ dtm\ [\![tff]\!]\ q\ t\ 0 = q \\ dtm\ [\![tff]\!]\ q\ t\ n = dtm\ [\![tff]\!]\ q\ t\ (n-1) \oplus t\ (n-1) \end{array}$$

we observe that:

$$toggle = dtm\ [\![tff]\!]\ 0$$

This leads to the following Glass description:

$$\begin{array}{ll}\textbf{def} & toggle \in E \Rightarrow E; \\ & \quad toggle\ e = tff\ 0\ e\end{array}$$

which has as discrete time behaviour:

$$dtm\ [\![toggle]\!] = toggle$$

We could also implement *toggle* by using a JK flipflop which has as discrete time behaviour:

$$\begin{array}{l} dtm\ [\![jkff]\!] : \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^2 \to (\mathbb{N} \to \mathbb{B}) \\ dtm\ [\![jkff]\!]\ q\ (j,k)\ 0 = q \\ dtm\ [\![jkff]\!]\ q\ (j,k)\ n = dtm\ [\![jkff]\!]\ q\ (j,k)\ (n-1)?\neg\ (k\ (n-1)); j\ (n-1) \end{array}$$

or written in a different way:

$$\begin{array}{l} dtm\ [\![jkff]\!] : \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^2 \to (\mathbb{N} \to \mathbb{B}) \\ dtm\ [\![jkff]\!]\ q\ (j,k)\ 0 = q \\ dtm\ [\![jkff]\!]\ q\ (j,k)\ n = \neg q' \wedge (j\ (n-1)) \vee q' \wedge \neg(k\ (n-1)) \\ \quad \textbf{where}\ q' = dtm\ [\![jkff]\!]\ q\ (j,k)\ (n-1) \end{array}$$

we see that *toggle* may be realized by using a *jkff* whose j and k input are connected together and whose initial state is 0, leading to the following Glass description:

$$\begin{array}{ll}\textbf{def} & toggle2 \in E \Rightarrow E; \\ & \quad toggle2\ e = jkff\ 0\ \langle e, e\rangle;\end{array}$$

which also implements the wanted discrete time behaviour:

$$dtm\ [\![toggle2]\!] = toggle$$

7.3.4 The final derivation

We define a generic *and* whose instances can be realized by gates:

$$\begin{array}{l}\textbf{def}\quad g\_and : \mathbb{N} \ni n \to \mathbb{B}^n \to \mathbb{B} \\ \textbf{with}\ g\_and\ 0\ \varepsilon = 1 \\ \qquad g\_and\ n\ (a \succ x) = a \wedge g\_and\ (n-1)\ x\end{array}$$

Proposition 7.17 $\forall(n, x : \mathbb{N} \ni n \times \mathbb{B}^n \,.\, g\_and\ n\ x = \forall(i : \Box n \,.\, x\ i))$

whose proof is trivial. Remark also that:

$$\begin{array}{l}g\_and\ 2 = s\_and2 = simplex\ [\![and2]\!] \\ g\_and\ 3 = s\_and3 = simplex\ [\![and3]\!]\end{array}$$

and that by theorem 7.12:

$$\begin{array}{l}(g\_and\ 2)^{\succ} = dtm\ [\![and2]\!] \\ (g\_and\ 3)^{\succ} = dtm\ [\![and3]\!]\end{array}$$

and so on.

The following proposition enables us to rewrite the definition of c_{Ctr} in terms of this generic and:

Proposition 7.18

$$\begin{array}{l}\forall\ (n, en, i : \mathbb{N} \ni n \times (\mathbb{N} \to \mathbb{B}) \times \Box n \,.\, en\ (t-1) \wedge \forall(j : \Box i \,.\, c_{Ctr}\ n\ en\ j\ (t-1)) = \\ \qquad (g\_and\ (i+1))^{\succ}\ (en \succ (j \in \Box i \,.\, c_{Ctr}\ n\ en\ j))\ (t-1))\end{array}$$

Proof

$$\begin{array}{lll} & & en\ (t-1) \wedge \forall(j : \Box i \,.\, c_{Ctr}\ n\ en\ j\ (t-1)) \\ = & \{\text{prop } 7.17\} & en\ (t-1) \wedge g\_and\ i\ (j : \Box i\ .\ c_{Ctr}\ n\ en\ j\ (t-1)) \\ = & \{\text{def } g\ and\} & g\ and\ (i+1)\ (en\ (t-1) \succ (j : \Box i\ .\ c_{Ctr}\ n\ en\ j\ (t-1))) \\ = & \{\text{prop } T\} & g\ and\ (i+1)\ (en\ (t-1) \succ T\ (j : \Box i\ .\ c_{Ctr}\ n\ en\ j)\ (t-1)) \\ = & \{\text{prop } T\} & g\ and\ (i+1)\ (T\ (en \succ (j : \Box i\ .\ c_{Ctr}\ n\ en\ j))\ (t-1)) \\ = & \{\text{def }{}^{\succ}\} & (g\_and\ (i+1))^{\succ}\ (en \succ (j : \Box i\ .\ c_{Ctr}\ n\ en\ j))\ (t-1)\end{array}$$

■

We subsitute this result into the definition of c_{Ctr}:

$$\begin{array}{l}\textbf{def}\ \ c_{Ctr} : \mathbb{N} \ni n \to (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})^n \\ \textbf{with}\ c_{Ctr}\ n\ en\ i\ 0 = 0 \\ \qquad c_{Ctr}\ n\ en\ i\ t = c_{Ctr}\ n\ en\ i\ (t-1) \oplus \\ \qquad\qquad ((g\_and\ (i+1))^{>}\ (en \gg (j : \Box i\ .\ c_{Ctr}\ n\ en\ j))\ (t-1))\end{array}$$

Next we substitute the subsystem of the previous subsection into this definition:

$$\begin{array}{l}\textbf{def}\ \ c_{Ctr} : \mathbb{N} \ni n \to (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})^n \\ \textbf{with}\ c_{Ctr}\ n\ en\ i = toggle\ ((g\_and\ (i+1))^{>}\ (en \gg (j : \Box i\ .\ c_{Ctr}\ n\ en\ j)))\end{array}$$

At this point we step to a concrete realisation. We must therefore fix the parameter n. Suppose that we want a 4 bit counter, so we substitute $n = 4$. We will also need individual identifiers for the 4 different signals. We will name the identifiers q_i, which will be equal to the signal $c_{Ctr}\ n\ en\ i$. We also substitute $dtm\ [\![toggle]\!]$ for *toggle*. We then obtain:

$$\begin{array}{l}\textbf{def}\ \ c_{Ctr4} : (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})^4 \\ \textbf{with}\ c_{Ctr4}\ en = (q_0, q_1, q_2, q_3) \\ \qquad \textbf{where} \\ \qquad\qquad q_0 = (dtm\ [\![toggle]\!])\ ((g\_and\ 1)^{>}\ (\tau\ en)) \\ \qquad\qquad q_1 = (dtm\ [\![toggle]\!])\ ((g\_and\ 2)^{>}\ (en, q_0)) \\ \qquad\qquad q_2 = (dtm\ [\![toggle]\!])\ ((g\_and\ 3)^{>}\ (en, q_0, q_1)) \\ \qquad\qquad q_3 = (dtm\ [\![toggle]\!])\ ((g\_and\ 4)^{>}\ (en, q_0, q_1, q_2))\end{array}$$

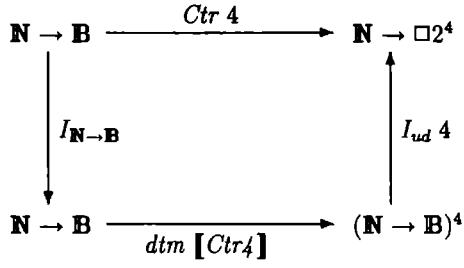
or substituting further:

$$\begin{array}{l}\textbf{def}\ \ c_{Ctr4} : (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})^4 \\ \textbf{with}\ c_{Ctr4}\ en = (q_0, q_1, q_2, q_3) \\ \qquad \textbf{where} \\ \qquad\qquad q_0 = (dtm\ [\![toggle]\!])\ en \\ \qquad\qquad q_1 = (dtm\ [\![toggle]\!])\ ((dtm\ [\![and2]\!])\ (en, q_0)) \\ \qquad\qquad q_2 = (dtm\ [\![toggle]\!])\ ((dtm\ [\![and3]\!])\ (en, q_0, q_1)) \\ \qquad\qquad q_3 = (dtm\ [\![toggle]\!])\ ((dtm\ [\![and4]\!])\ (en, q_0, q_1, q_2))\end{array}$$

Inverting *dtm* yields the following Glass definition:

$$\begin{array}{l}\textbf{def}\ \ Ctr4 \in E \Rightarrow E^4; \\ \qquad Ctr4\ en = \langle q_0, q_1, q_2, q_3 \rangle \\ \qquad \textbf{where} \\ \qquad\qquad q_0 = toggle\ en; \\ \qquad\qquad q_1 = toggle\ \langle and2\ \langle en, q_0 \rangle \rangle; \\ \qquad\qquad q_2 = toggle\ \langle and3\ \langle en, q_0, q_1 \rangle \rangle; \\ \qquad\qquad q_3 = toggle\ \langle and4\ \langle en, q_0, q_1, q_2 \rangle \rangle; \\ \qquad \textbf{endwhere}\end{array}$$

and we have proven that the following diagram indeed commutes:



7.3.5 The complete Glass description

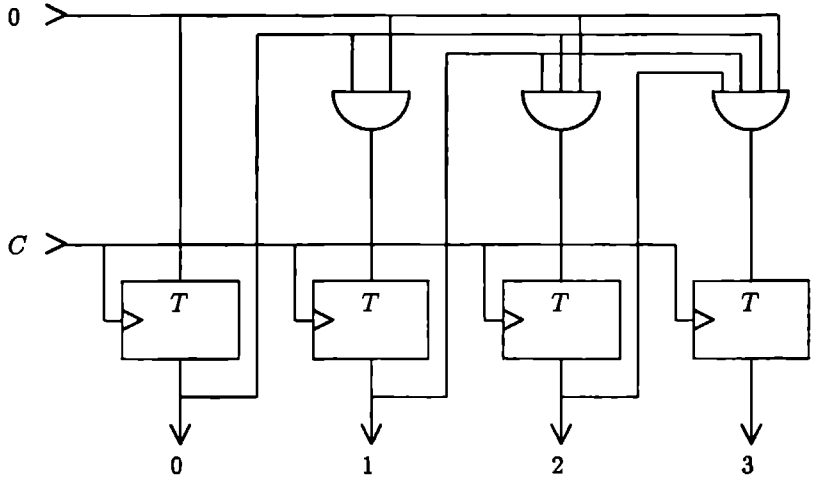
basetype E

atom $and2 \in E^2 \Rightarrow E$,
$and3 \in E^3 \Rightarrow E$,
$and4 \in E^4 \Rightarrow E$,
$tff \in \mathbb{B} \to E \Rightarrow E$;

def $toggle \in E \Rightarrow E$;
$toggle\ en = tff\ 0\ en$;

def $Ctr4 \in E \Rightarrow E^4$;
$Ctr4\ in = \langle q_0, q_1, q_2, q_3 \rangle$
where
$q_0 = toggle\ in$;
$q_1 = toggle\ \langle and2\ \langle in, q_0 \rangle \rangle$;
$q_2 = toggle\ \langle and3\ \langle in, q_0, q_1 \rangle \rangle$;
$q_3 = toggle\ \langle and4\ \langle in, q_0, q_1, q_2 \rangle \rangle$;
endwhere

This description has the following structural interpretation:



7.4 Conclusions

Two examples of a hardware design approach were given. The idea behind this approach is that a hardware design may be derived by proving its properties. An analogy exists in a software design approach, in which software is derived by setting out from a high level formal specification, which is transformed step by step into a specification which is executable, i.e. a program. We believe that our approach should be practised more often in current VLSI design.

Chapter 8

Considerations regarding asynchronous circuits

And it is said by the Eldar that in water there
lives yet the echo of the Music of the Ainur
more than in any substance else that us in this Earth;
and many of the children of Iluvatar hearken still
unsated to the voices of the sea, and yet know not
for what they listen.
Ainulindalë *J.R.R. Tolkien*

8.1 Introduction

In this chapter we will discuss some (but certainly not all) models which can be used in implementing behavioural semantic functions for the complete directional subset of Glass. Since this topic is highly complex, dealing with arbitrary interconnection and feedback in digital networks, research in this area could easily constitute a thesis in itself. We will therefore limit ourselves and discuss several approaches for implementing these semantic functions. Thus this chapter is more descriptive and less formal than the previous ones. Some semantic functions are also described in a nonfunctional way.

This exercise also illustrates one of the advantages of Glass: to study another behavioural model of a digital network one does not rewrite the Glass description of the network, but merely applies another semantic function to the unchanged description (assuming of course that other semantic functions are available).

We can consider two approaches to study the behaviour of asynchronous hardware. One approach is to derive certain (safety) properties of the system based upon a certain race or hazard model. The other approach is to simulate the hardware according to some time model.

8.2 Race and hazard models

8.2.1 Four representative models

Of the first approach we mention the following four:

- the Feedback delay model of Huffman [Huf54].
- the Gate delay model of Muller and Bartky [Mul59]
- the Ternary Simulation model of Eichelberger [Eic65]
- Formal Race models of Brzozowski et al. [Brz75]

One common trait of these models is that they all assume *fundamental mode operation.* By fundamental mode operation is meant that the behaviour of a circuit is observed by starting the circuit in a stable state, changing the input vector to a new value and then waiting for the final outcome of this change of input vector, allowing the circuit to completely settle before the inputs are allowed to change again. Behaviour due to input changes while the circuit is still settling is therefore not considered in these models.

The first two of the above mentioned models differ only in the location of the delays. Unfortunately these two models are too optimistic. It is possible to construct circuit (for instance Langdon's example) whose real behaviour can not be explained by the model. At present the most accurate model seems to be the gates with delay and wires with delay model.

In 1975 Brzozowksi et al. developed formal race models (The General Multiple Winner model (GMW model)) for analyzing asynchronous circuits. Their approach consists of considering all possible outcomes of a transition. Their models are very pessimistic (The delay in a wire may change from one moment to another, etc). However, if a transition is safe in this model (having only a single outcome) it is also safe in reality.

In 1987 Brzozowski and Seger proved that the results of ternary simulation and those provided by the General Multiple Winner model were the same, thus providing an easy algorithm for this model.

8.2.2 Ternary simulation

Ternary simulation of an input transition of a network consists of applying two algorithms. In these algorithms nodes can take values in $\mathbf{E} = \{0, \perp, 1\}$ (see also section 4.4). Both algorithms need considering the possible values of all internal nodes (including the input nodes). For this reason the input vector is extended into a state vector consisting of the values of all nodes. For an input transition from x into x' the following two algorithms are executed.

First the input vector is changed from x into x'' by replacing all those values in x that change in the transition by $\perp$. This may trigger the changing of the values on some of the internal nodes in the network according to the definitions given for the atoms in section 4.4.

We can define a partial order on $\mathbf{E}$ by restricting the lattice of section 4.5 to $\mathbf{E}$. Remark that we may read this partial order relation as "is less defined than". This partial order can be extended into a partial order for arrays of values in $\mathbf{E}$. One can easily show that the definitions given in section 4.4 are monotonous with respect to this partial order. For instance:

$$(a, b) \sqsubseteq (c, d) \Rightarrow e\_and\ (a, b) \sqsubseteq e\_and\ (c, d)$$

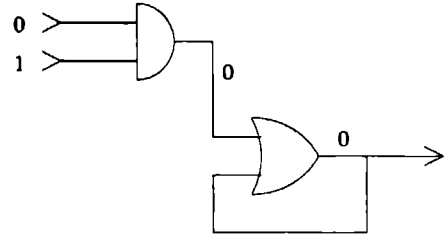
Consequently, the values of the internal nodes of the network will change into less defined values until a (unique) fixed point is reached. Thus this fixed point is easily calculated by the following algorithm:

WHILE there is an internal node triggered to change its value
DO set the value of this node to $\perp$ **OD**

The finiteness of the network ensures the termination of this algorithm.

Next the input vector is changed to its final value. Again a fixed point is sought in much the same way as above with the exception that the nodes in the network will now strife to more defined values instead of less defined ones. If the state vector resulting from this second algorithm still contains $\perp$ values, the transition is not safe in the GMW model.

As an example, consider the following circuit, initially configured as indicated by the labels.



Consider an input transition from 01 to 10. Remark that in the gate delay model the output will never attain the value 1. Ternary simulation, however, will show that this transition is not safe. The first algorithm will show the following sequence of transitions:

$$0100 \rightarrow \perp\perp00 \rightarrow \perp\perp\perp0 \rightarrow \perp\perp\perp\perp$$

The second will give:

$$\perp\perp\perp\perp \rightarrow 10\perp\perp \rightarrow 100\perp$$

As we can see, a fixed point is reached while the output remains undefined. This is, however, a correct observation since the output may attain the value 1, if the delay in the lower input wire of the And gate is much larger than the delay in the upper one.

One possible implementation of a semantic function based upon this model is the following: Firstly, the Glass description of the system is completely flattened until all internal nodes have unique names, thus enabling a state vector to be formed, followed by hardcoding all atom applications into an executable. Using this code the above algorithms could be used to flag all safe and unsafe input transitions.

8.3 Simulation of hardware

As indicated earlier, many of the formal models are either too optimistic (not accurate enough) or too pessimistic (allowing for races that will never occur in reality). Simulation of the hardware may be a viable alternative especially if the technology with which the hardware will be realized is incorporated in the simulation. We will consider three possible simulation models:

- Fixed delay simulation
- Models based on functionals.
- Discrete event simulation

8.3.1 First approximation: fixed delay model

A simple model for asynchronous feedback can be obtained from the discrete time function model by giving the gates a nonzero delay:

Def

$$u\ not \in (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B} \qquad \tau_A\ [\![not]\!] = u\_not$$
$$u\ not\ x\ n = s\_not\ (x\ (n-1))$$

Def

$$u\ and \in (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B} \qquad \tau_A\ [\![and]\!] = u\_and$$
$$u\ and\ (x,y)\ n = s\_and\ (x\ (n-1), y\ (n-1))$$

and so on. This model still conforms to the generic model, formulated in chapter 6. Thus the application of this semantic function (denoted by *utm* (unit delay timing model)), to the following Glass description of a RSFF flipflop

```
Def
    RSFF ∈ E² ⇒ E²;
    RSFF ⟨r, s⟩ = ⟨q, q'⟩
        where
            q = nand ⟨s, q'⟩;
            q' = nand ⟨r, q⟩;
        endwhere
```

yields:

```
utm [rsff] : (N → B)² → (N → B)²
utm [rsff] (r, s) = (q, q')
    where
        q = u_nand (s, q')
        q' = u_nand (r, q)
```

The following table shows a typical simulation run obtained from the previous definition:

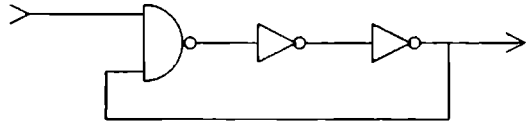
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
r	0	0	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1
s	1	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1
q	0	1	0	0	0	0	0	0	1	1	1	1	1	0	1	0	1
q'	0	1	1	1	1	1	1	1	1	0	0	1	1	0	1	0	1

Notice that, in this model, the feedback loops need two steps in time before they settle. We can also observe oscillating behaviour due to the transition $(r, s) = (0, 0)$ to $(r, s) = (1, 1)$.

The advantage of this model is that as long as the delays of gates are fixed and that rise and fall times of a gate are equal, the efficient algorithm given in chapter 6 can still be used. The disadvantage is that the above conditions are actually never met in real hardware. In many technologies the rise time of a gate is larger then its fall time (Oft this is due to the fact that holes in P material have a lower mobility than electrons in N material. It may also be caused by certain ratio requirements such as exist in NMOS technologies). The model also shows some deficiencies of its own. Consider for instance the following Glass description of a ring oscillator:

```
def  ring ∈ E ⇒ E³;
     ring a = ⟨x, y, z⟩
         where
           x = nand ⟨a, z⟩;
           y = not x;
           z = not y;
         endwhere
```

which has the following structural interpretation:



Applying the semantic function to this definition yields:

$$
\begin{array}{l}
utm\ [\![ring]\!] : (\mathbf{N} \to \mathbf{B}) \to (\mathbf{N} \to \mathbf{B})^3 \\
utm\ [\![ring]\!]\ a = (x, y, z) \\
\quad \textbf{where} \\
\qquad x = u\_nand\ (a, z) \\
\qquad y = u\_not\ x \\
\qquad z = u\_not\ y
\end{array}
$$

If we then simulate the circuit using this definition, we obtain the following output:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
x	0	1	0	1	0	1	1	1	0	0	0	1	1	1	0	0	0	1
y	0	1	0	1	0	1	0	0	0	1	1	1	0	0	0	1	1	1
z	0	1	0	1	0	1	0	1	1	1	0	0	0	1	1	1	0	0

As we can see the initial oscillation is entirely due to the model, while the later oscillation is the one we would ordinarily expect.

8.3.2 Models based on functionals

One possible way to go is to define suitable higher order functions that model real gates in some technology and use these definitions as the properties of the atoms. In this section we will introduce a functional model that exhibits the following characteristics:

- The rise of an output is twice that of its fall time (which is not unusual in CMOS, unless the designer compensates the lower mobility of the holes by making his P transistors twice as wide as his N transistors).
- During a transition, the output may have an undefined value.
- If an input of a gate has undefined value (i.e. is in transition), the output will attain undefined value too.

Our model will still be based upon the generic model presented in chapter 6, although it can no longer be implemented in an efficient way. As usual, we define the properties of the atoms. The actual domain of interpretation is $\mathcal{D} = \mathbf{N} \to \mathbf{E}$.

$$
\begin{array}{l}
\textbf{def}\ \ —t : (\mathbf{N} \to \mathbf{E}) \to (\mathbf{Z} \to \mathbf{E}) \\
\textbf{with}\ x'\ t = (t < 0)?0 \natural x\ t
\end{array}
$$

$$\begin{array}{l}\textbf{def}\ \ f\_not : (\mathbb{N} \to \mathbb{E}) \to (\mathbb{N} \to \mathbb{E}) \\ \textbf{with}\ f\_not\ x\ n = \ (x'\ (n-1) = 1)?0\dagger \\ \qquad\qquad\qquad\quad (x'\ (n-1) = 0 \wedge x'\ (n-2) = 0)?1\dagger \\ \qquad\qquad\qquad\quad \bot\end{array}$$

$$\begin{array}{l}\textbf{def}\ \ f\_nand : (\mathbb{N} \to \mathbb{E})^2 \to (\mathbb{N} \to \mathbb{E}) \\ \textbf{with}\ f\_nand\ (x,y)\ n = (x'\ (n-1) = 1 \wedge y'\ (n-1) = 1)?0\dagger \\ \qquad\qquad\qquad\qquad\quad ((x'\ (n-1) = 0 \wedge x'\ (n-2) = 0) \vee \\ \qquad\qquad\qquad\qquad\quad (y'\ (n-1) = 0 \wedge y'\ (n-2) = 0))?1\dagger \\ \qquad\qquad\qquad\qquad\quad \bot\end{array}$$

If we apply the semantic function belonging to these properties to the description of the RS flipflop and then simulate using the resulting function we obtain:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
r	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1
s	1	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1
q	1	0	0	0	0	0	0	0	$\bot$	1	1	1	1	1	0	$\bot$	$\bot$	$\bot$
q'	1	1	1	1	1	1	1	1	1	$\bot$	0	0	$\bot$	1	0	$\bot$	$\bot$	$\bot$

As we can see the final output remains undefined, which we would expect beforehand.

These functional models deserve much more research. They would gain in power if we could take continuous domains such as $\mathbb{N} \to \mathbb{R}$ or even $\mathbb{R} \to \mathbb{R}$ as domain of interpretation.

8.3.3 Discrete event simulation

Discrete event simulation is a third way to go. An event is the changing of the values on one of the nodes in the network. While simulating an event queue is maintained while keeping the queue ordered in time. In every step of the simulation an event is dequeued from the head of the queue and the value of the corresponding node in the network is updated. Next for all gates connected to this node is calculated whether this transition will cause a transition on the output of the gate. If so these output event are enqueued on the event queue. Simulation will proceed until no events remain on the event queue (which by the way may never become empty).

As we can see discrete event simulation also needs names for every internal node in the network. Hence a Glass description must be completely flattened before we can generate a discrete event simulator for it. The author has implemented such a flattener and discrete event simulator generator. If we apply this semantic function to the Glass description of the ring oscillator and then execute the generated simulator we can obtain the simulation run presented in the figure on the next page. The propagation times are those as typically specified for Schottky TTL.

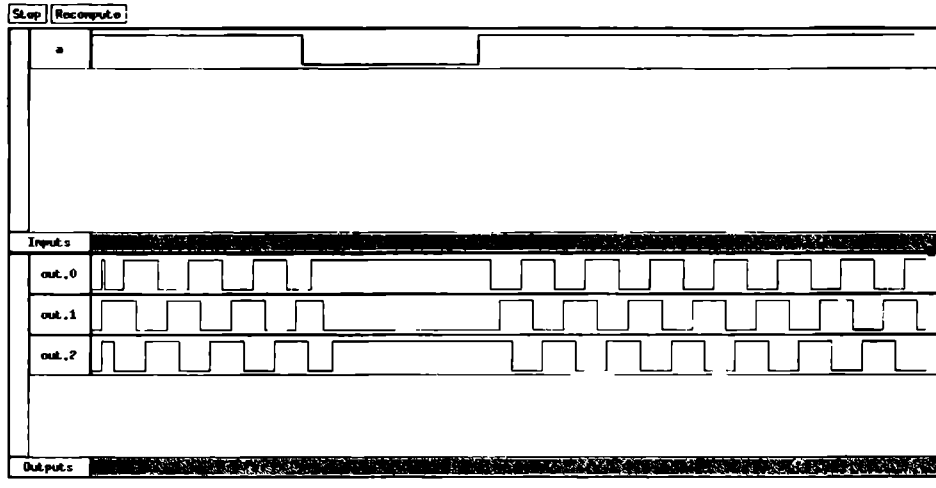


Figure 8.1: Xring screendump

8.4 Conclusions

In this chapter we only glimpsed at ways to implement semantic functions that can handle asynchronous hardware. Much research is still necessary especially in developing sound functional models for gates.

Chapter 9

VHDL versus Glass

Ents, the earthborn, old as mountains,
the wide-walkers, water drinking;
and hungry as hunters, the Hobbit children,
the laughing-folk, the little people.
The Lord of the Rings *J.R.R. Tolkien*

9.1 VHDL

9.1.1 Introduction

VHDL is a language for describing digital electronic circuits. In 1980 the U.S. Department of Defense (DoD) initiated the VHSIC (Very High Speed Integrated Circuits) program. During its course the need arose for a language for describing the structure and behaviour of integrated circuits. The VHDL (VHSIC Hardware Description Language) language was therefore developed and soon adopted as IEEE standard 1076 [IEEE88]. Moreover, the DoD has prescribed a particular validation set to which VHDL tools have to comply.

VHDL describes two aspects of a design, namely its structure and (a simulation of) its behaviour. The latter is specified by means of a discrete event simulation.

The VHDL syntax closely resembles the syntax of existing programming languages, especially that of Ada. Because of this reason the syntax of VHDL is as baroque as that of Ada. We will therefore not discuss the exact syntax of VHDL. In a VHDL design environment a description of a design may be executed (simulated) so that designers can compare different alternatives without the expense of hardware prototyping.

9.1.2 Entities

The structure of a digital system is often described as a hierarchical collection of modules. Each module has a set of terminals which constitute its interface to the outside world. In VHDL terminology, such a module is called an *entity*. An entity

represents a part or whole of a hardware design, having a well defined interface and a well defined behaviour.

An *entity declaration* introduces a name for an entity. It also serves to introduce the external interface of the entity. The terminals of an entity are called *ports* in VHDL. Optionally subcomponents may be declared in the entity declaration. In most cases, however these subcomponents are only described in the implementation of the entity and not in its declaration. In such cases the entity declaration gives the outside black box view of the entity: only its name and its external interface are introduced. An example of such an entity declaration is the following declaration of a processor:

```
entity processor is
    generic (max_freq: frequency := 16 MHz);
    port (clock: in bit;
          address: out integer;
          data: inout word_32;
          control: out proc_control;
          ready: in bit);
end processor;
```

Remark that certain behavioural aspects of the design are already present in this declaration namely in the specification of the types of the ports.

Objects in VHDL are characterized by their class, type and value. There are three object classes namely constants, variables and signals. The usage of constants and variables is quite analogue to that in ordinary (imperative) programming languages. *Signals* are used to connect submodules in a design. When a signal is associated with a component or module a relative direction must also be specified. Such a direction is one of the following five: **in** (input), **out** (output), **inout** (bidirectional input/output), **buffer** or **linkage** (any direction). In the examples above the **port** clause introduces the signals associated with the external interface of the entity 'processor'.

In this example the generic constant can be used by the behavioural description of the entity to specify signal delays. Generic parameters may be used to specify classes of entities with a varying structure. Consider for instance the following entity declaration of a ROM with varying sizing:

```
entity ROM is
    generic (width, depth: positive);
    port (enable: in bit;
          address: in bit vector(depth - 1 downto 0);
          data: out bit_vector(width - 1 downto 0));
end ROM;
```

When such a parametrized entity is used as a component its generic parameters must be instantiated.

9.1.3 Describing structure in VHDL

One or more implementations (aspects) of an entity may be declared in so-called *architecture bodies*. Such an architecture body has two parts namely a declarative part and a statement part. The declarative part serves to introduce additional types, constants, variables, components, subprograms and signals.

VHDL views components as submodules that are described separately in a design library. A component declaration incorporates a component in an architecture body. An example of such a declaration is the following:

```
component or
    port (a,b: in bit, y: out bit);
end component
```

Components may have generic parameters which may be used for instance to specify propagation delays or sizing information.

The statement part of an architecture body serves to introduce component instantiations and *blocks*. Blocks specify submodules of an architecture body. A block is a complete module with its own external interface, connected to other blocks or components by signals. Within a block new (internal) signals, components, may be declared in the declarative part of the block and component instantiations and (sub) blocks in its statement part (i.e. blocks may be nested). In this way architecture bodies can be decomposed hierarchically.

Component instantiations and blocks must be given a unique label. In this way every component and every block in a design has a unique name. Associated with a component instantiation or a block is a **port map** clause which specifies the connection of the interface signals of the submodule to the interior signals of the surrounding architecture body or block. For instance the above declared 'or' can be instantiated as follows:

```
or_33: or port map (a ⇒ im_1, b ⇒ im_2, y ⇒ out_42);
```

The reader should be aware that the arrow ⇒ does not indicate direction but merely indicates signal binding. If a component has generic parameters the instantiation of this component must also specify concrete values for these parameters by means of a **generic map** clause. This latter may be omitted if the component declaration already specified default values for these parameters.

An example of a block can be found in the following (prototype) structural description of the example entity 'processor'.

```
architecture block_structure of processor is
    declarations
    type datapath_control = .....
    signal internal_control: datapath_control
begin
```

```
    control_unit: block
        port (clk: in bit,
                bus_control: out proc_control,
                bus_ready: in bit,
                control: out datapath_control);

        port map (clk ⇒ clock,
                bus_control ⇒ control,
                bus_ready ⇒ ready,
                control ⇒ internal_control);

        other declarations for control_unit
        begin
                statements for control_unit
        end block control_unit;
    other statements for block structure
end block_structure;
```

9.1.4 The structural description of a data selector in VHDL

Using the constructs introduced in the previous subsections we can give a structural description of the data selector in VHDL:

```
entity select is
    port (sel: in bit;
            a,b: in bit;
            out: out bit);
end select;

architecture structure of select is
component not port (a: in bit; y: out bit);
end component;

component and port (a,b: in bit; y: out bit);
end component;

component or port (a,b: in bit; y: out bit);
end component;

signal sel_bar: bit;
signal im_a: bit;
signal im_b: bit;

begin
    inv0: not port map (a ⇒ sel, y ⇒ sel_bar);
```

```
    and0: and port map (a ⇒ sel bar, b ⇒ a, y ⇒ im_a);
    and1: and port map (a ⇒ sel, b ⇒ b, y ⇒ im_b);
    or0: or port map (a ⇒ im_a, b ⇒ im_b, y ⇒ out);
end structure;
```

Like in the entity declaration there are behavioural aspects in this "structural" description, namely in the type specification of the signals.

9.1.5 Describing behaviour in VHDL

The behaviour of a VHDL entity is specified by a discrete event time model of the entity, which can be executed.

At some simulation time an input may be stimulated by changing its value. The entity will then react by running the code of its behavioural description and scheduling new values to be placed on signals connected to the output ports or on internal signals at some later simulated time. In VHDL terminology this is called scheduling a *transaction* on that signal. If the new value is different from the old value of the signal, an *event* occurs which may activate other modules via input ports connected to that signal.

Simulation starts with an *initialisation* phase in which all signals get their initial values and simulation time is set to zero. Simulation then proceeds with a two stage *simulation cycle*. In the first stage the simulation time is advanced to the earliest time at which a transaction is scheduled. All transactions scheduled for that time are then executed causing events to occur on some signals. In the second stage all modules reacting to events occurring in the first stage then execute their behavioural programs, usually scheduling further transactions. When all behavioural programs have finished executing, the cycle repeats. This continues until there are no more scheduled transactions.

A signal assignment schedules one or more transactions to a signal. One therefore associates with each signal a so called *projected waveform*. For example if the statement

```
s ⇐ '0' after 10ns
```

was executed at simulation time 5ns this transaction will be processed at simulation time 15ns. If a signal assignment is executed and there are already transactions scheduled at a later simulation time, these are deleted. Depending on the kind of delay specified in the assignment (transport or inertial delay) transactions occurring earlier than the new ones scheduled may also be deleted. If the keyword **after** is omitted the signal assignment is executed immediately (i.e. with a zero delay).

Discrete event models of an entity are described by a set of concurrently executing statements such as signal assignments, *processes*, etc. A process is a sequential piece of code, which can be activated as a response to certain signal transactions, thereby possibly scheduling new transactions. Different processes may be active at the same time. Like blocks, processes may be labeled.

Processes are activated initially by the initialisation phase of the simulation. When active they repeatedly execute their sequential statements until they suspend themselves by executing a **wait** statement. Associated with a **wait** statement is a *sensitivity list* of signals to which the process is sensitive during its suspension. When a transaction occurs on one of the mentioned signals and the new values of the signals satisfy the condition specified in the **wait** statement the process will resume. This condition is optional. An example of a process is the following specification of a Müller C element:

```
process
begin
    wait until a = '1' and b = '1';
    q ⇐ '1' after Tprop;
    wait until a = '0' and b = '0';
    q ⇐ '0' after Tprop;
end process
```

A sensitivity list may also be specified in the header of the process. The process then executes an implicit **wait** statement prior to its sequential statements. Variables may be used to store some state of the process. Consider for instance the following description of the behaviour of a (synchronous) D flipflop:

```
process (clock)
begin
    if clock = '1'
    then
        q ⇐ d after prop_delay
    end if
end process;
```

9.1.6 The behavioural description of a data selector in VHDL

Using the above constructs we can give a behavioural description of our data selector in VHDL:

```
architecture behaviour of select is
begin
    sel: process (s,a,b)
            out ⇐ (not s and a) or (s and b) after 10ns;
        end process;
end behaviour;
```

9.2 VHDL and Glass

There are some common points in both of the languages:

- In both languages one describes a circuit to study its behaviour and other aspects without resorting to hardware prototyping.
- Both languages support hierarchy: one can describe the decomposition of a system in subsystems in both languages. In such a description VHDL shows some behavioural aspects.
- Both languages are strongly typed. Both use typing as part of the interface specification.

9.3 VHDL contra Glass

- A Glass description of a system precisely describes the decomposition of a system and the interconnection of its subsystems: it therefore gives a perfect black box model of the system. This is impossible to do in VHDL, since there are always behavioural aspects in a VHDL description such as in the typing of the interface.
- The syntax of Glass is easy to read. This is in contrast with the baroque of VHDL. Glass descriptions are also more concise by their elegance.
- Glass aims at notational economy: one does not need to change the description of a system in order to try a new model of that system. One merely applies another semantic function to this description.
- When writing a behavioural description in VHDL, one actually specifies a simulator. When deriving the behaviour of a system described in Glass, one sometimes derives a simulator.
- The discrete event simulation model is the lowermost level model available in VHDL to describe behaviour. When using Glass, one does not rule out the possibility of mixed mode simulation or analog simulation.
- When a user describes two architecture bodies for one design entity in VHDL, one can always doubt the consistency between these two descriptions. For a system described in Glass, one derives different semantic models from one description only.
- The correctness of a semantic function (i.e. the correctness of a certain model) are the sole responsibility of the implementor of the semantic function. The implementation of a new model takes a considerable amount of time.

9.4 Conclusions

As we have ostensibly demonstrated, VHDL system descriptions never escape fully from certain behavioural aspects. The discrete event model chosen for VHDL restricts the

esigner also in the level with which he can observe the behaviour of the system und
esign. Moreover this is the only behavioural model available in VHDL. Should a us
ant a lower level behavioural model she must resort to other tools to whose inpu
rmat the VHDL description must be translated or rewritten. Apart from this proble
behavioural and a structural description of one design entity need not be consisten
he author therefore believes that the usage of VHDL in hardware description shoul
e discouraged.

Chapter 10

Final remarks

The Road goes ever on and on
Out from the door where it began
Now far ahead the Road has gone
Let others follow it who can
Let them a journey new begin
But I at last with weary feet
Will turn towards the lighted inn
My evening-rest and sleep to meet
Lord of the Rings *J.R.R. Tolkie*

s we have seen much research is still necessary into the field of systems semanti pecially where continuous time and value models are concerned. We hope that futu searchers can develop more theory in this direction. As we have seen in chapter ew models will inevitably give rise to descriptions involving higher order function esearch is therefore also necessary to develop really efficient implementations of fun onal languages.

Appendix A

Kernel language datastructure definition file

```
|| File: nkerngl.ds
|| Author: C. van Reeuwijk

orig == (file:string, line:inum);

|| Definitions.
def ::=
    DefAtom atorig:orig atnm:symbol atptyp:[partyp] atctyp:typ | || atom
    DefBasetype baseorig:orig basename:symbol | || basetype
    DefVal valorig:orig valnm:symbol valtyp:typ valas:val | || def
    DefCon conorig:orig defcon:val conas:val  | || defcon = conas
    DefTyp typorig:orig typnm:symbol typas:typ ; || type

|| Possible types of parameters types
partyp ::=
    PTInt |
    PTFlo |
    PTStr |
    PTBool ;

|| Type expressions
typ ::=
    TypBase basenm:symbol | || basetype
    TypIn ityp:typ | || ?ityp
    TypOut otyp:typ | || !otyp
    TypUni uityp:typ uotyp:typ | || uityp => uotyp
    TypNon nontyp:typ | || non-directional system
    TypProd ptypes:[typ] | || Carthesian product.
```

```
    TypSym sym:symbol; || symbolic type

|| Formal connection patterns.
formcon ::=
    FCList l:[formcon] | || list of formal connections
    FCSym sym:symbol; || formal connection symbol

|| parameters of parameterized atoms
parval ::=
    ParInt i:inum | || int
    ParFlo f:fnum | || float
    ParStr s:string | || string
    ParBool b:bool ; || bool

|| Value expressions.
|| Atoms and apply-s have a source file and line.
val ::=
    VSym symorig:orig sym:symbol | || Symbol
    VLambda lpar:formcon lval:val | || Lambda abstraction
    VSigma spar:formcon sval:val | || Sigma abstraction
    VApply aval:val apar:val | || Apply (asval aspar)
    VWhere wdefs:[def] wval:val | || local context
    VList l:[val] | || List expression
    VAppset aps:[val] | || appset
    || Atom application
    VAtom atorig:orig atnm:symbol atvpar:[parval] atcpar:val |
    VSyn synlist:[val]; || Synonym expression
```

Appendix B

Implementation language equivalents

B.1 Equivalents for section 3.4

```
basetype E;

atom not :- E => E,
     and :- E & E => E,
     nand :- E & E => E,
     or :- E & E => E,
     xor :- E & E => E;

def same :- E => E;
    same x = not (not x);

def select :- E & E & E => E;
    select [s,a,b] = or [and [not s, a], and [s,b]];

def halfadder :- E & E => E & E;
    halfadder [x,y] = [xor [x,y], and [x,y]];

def ornot :- E & E => E;
    ornot = %[a,b].not (or [a,b]);

def SetOnce :- E => E;
    SetOnce i = o where o = or [i,o] endwhere;

def RSFF :- E & E => E & E;
    RSFF [R', S'] = [q, q']
```

```
    where
       q = nand [S', q'];
       q' = nand [R', q]
    endwhere;

def my_xor :- E & E => E;
    my_xor [x,y] = nand [nand [b,x], nand [b,y]]
       where b = nand [x,y] endwhere;

atom tbuf :- E & E => E,
     join4 :- E^4 => E;

def busif :- E^4 & E^2 => E;
    busif [[s0,s1,s2,s3],[a0,a1]] = join4 [t0,t1,t2,t3]
       where
          t0 = tbuf [nand [a0', a1'], s0];
          t1 = tbuf [nand [a0, a1'], s1];
          t2 = tbuf [nand [a0', a1], s2];
          t3 = tbuf [nand [a0, a1], s3];
          a0' = not a0;
          a1' = not a1;
       endwhere;
```

B.2 Equivalents for section 3.6

```
basetype E;

atom R :- [E & E],
     C :- [E & E],
     NPN :- [E & E & E],
     Xtal :- [E & E];

def RCnet :- [E & E & E];
    RCnet [a,b,c] = { R [a,c], C [b,c] };

def RCnet2 :- [E & E & E];
    RCnet2 [a,d,x] = { R [a,x], C [d, y], * [x,y] };

atom Supply :- [E],
     Gnd :- [E];

def VoltageFollower :- [E & E];
```

```
    VoltageFollower [in, out] =
       { NPN [out, in, plus],
         R [out, gnd],
         Supply plus,
         Gnd gnd
       };

atom Monoflop :- [?E & E & E & !E],
     not :- E => E;

def delay :- E => E;
    delay [in, out] = { Monoflop [in, x, y, out], C [x,y] };

def Xosc :- [!E];
    Xosc z =
       { R [x,y],
         R [y,z],
         Xtal [x,z],
         not [x,y],
         not [y,z]
       };

atom
   Nenh :- [E & E & E],
   Penh :- [E & E & E],
   Vdd :- [E];

def
   CMosInvertor :- E => E;
   CMosInvertor [in, out] =
      { Penh [in, plus, out],
        Nenh [in, gnd, out],
        Vdd plus,
        Gnd gnd
      };
```

B.3 Equivalents for section 3.7

```
basetype E;

mac triple :- E => E -> E => E;
    triple A in = A ( A ( A in));
```

```
atom R :- Float -> [E & E];

type tp = E & E => E;
```

B.4 Equivalents for section 3.8

```
atom divide_by_two :- E => E;

def divide_by_8 :- E => E;
    divide_by_8 c = triple divide_by_two c;

mac chain :- Int -> E => E -> E => E;
    chain n A in = n = 0 -> in; A (chain (n-1) A in);

def divide_by_1024 :- E => E;
    divide_by_1024 c = chain 10 divide_by_two c;

atom adc :- E & E & E => E & E;

mac
   nbitsadder :- Int -: n -> E^n & E^n & E => E & E^n;
   nbitsadder 0 [[],[], c] = [c, []];
   nbitsadder n [a:as, b:bs, cin] = [cout, s:ss]
      where
         [v,s] = adc [a,b,cin];
         [cout, ss] = nbitsadder (n-1) [as, bs, v];
      endwhere;

def Fourbitsadder :- E^4 & E^4 & E => E & E^4;
    Fourbitsadder [as,bs,cin] = nbitsadder 4 [as,bs,cin];

atom
   and5 :- E^5 => E,
   and4 :- E^4 => E,
   and3 :- E^3 => E,
   and2 :- E & E => E,
   nand :- E & E => E,
   nand3 :- E & E & E => E,
   or :- E & E => E,
   jkff :- E & E => E,
   buf :- E => E;
```

```
def
   counter :- E^4 & E & E & E & E => E^4 & E;
   counter [[d0,d1,d2,d3],nload,nclear,enap,enat] =
      [[q0,q1,q2,q3],carry]
      where
         q0 = counterstage [d0, preset, en, nclr];
         q1 = counterstage [d1, preset, and2 [q0,en], nclr];
         q2 = counterstage [d2, preset, and3 [q1,q0,en], nclr];
         q3 = counterstage [d3, preset, and4 [q2,q1,q0, en], nclr];
         carry = and5 [q3,q2,q1,q0,enat];
         preset = nand [nload, nclr];
         nclr = buf nclear;
         en = and2 [enap,enat];
         mac
            counterstage :- E^4 => E;
            counterstage [d, preset, toggle, nclr] =
               jkff [and2 [ja, ea], and2 [ka, ea]]
               where
                  ja = nand [ka, preset];
                  ka = nand3 [d, preset, nclr];
                  ea = or [toggle, preset]
               endwhere;
      endwhere;

atom odd :- E & E => E & E,
     even :- E & E => E & E;

mac row :- Int -> Int -: l -> E^l & E => E^l & E;
    row k 0 [[], e] = [[], e];
    row k l [n:N, e] = [s:S, w]
       where
          [s, ie] = (k = 0 -> even; odd) [n,e];
          [S, w] = row (1-k) (l-1) [N, ie];
       endwhere;

mac board :- Int -> Int -: l -> Int -: m -> E^l & E^m => E^l & E^m;
    board k l 0 [N, []] = [N, []];
    board k l m [N, e:E] = [S, w:W]
       where
          [IS, w] = row k l [N,e];
          [S, W] = board (1-k) l (m-1) [is,E];
       endwhere;
```

```
def chess :- E^8 & E^8 => E^8 & E^8;
    chess [N,E] = board 0 8 8 [N,E];

def smallchess :- E^4 & E^4 => E^4 & E^4;
    smallchess [N,E] = board 0 4 4 [N,E];
```

Bibliography

[Bou85] R.T. Boute,
"Formal description of arbitrary systems by means of functional languages",
Esprit project proposal 881, Technical annex,
December 1985

[Bou88] R.T. Boute,
"Systems Semantics: Principles, Applications, and Implementation."
ACM Transactions on Programming Languages and Systems,
Vol. 10, No 1, Jan 1988, pp 118-155

[Bou91] R.T. Boute,
"Functional and declarative formalisms",
Course notes OFD,
Autumn 1991

[Bou92] R.T. Boute,
"A declarative formalism supporting hardware/software co-design",
University of Nijmegene, 1992

[Brz75] J.A. Brzozowski and M. Yoeli,
"Models for Analysis of Races in Sequential Networks", in Lecture Notes in Computer Science, Vol 28, A. Blikle, ed., Springer Verlag, 1975, pp 26-32

[Eic65] E.B. Eichelberger,
"Hazard detection in combinational and sequential circuits",
IBM Journal Res. and Dev.,
March 1965, pp 90-99

[For92] T.E. Forster,
Set Theory with a Universal set,
Clarendon Press, Oxford, 1992

[Has90] M.R. Haskard,
"An Introduction to Application Specific Integrated Circuits",
Prentice Hall of Australia Pty Ltd., 1990

[Huf54] D.A. Huffman,
"The Synthesis of Sequential Switching Circuits",
J. Franklin Inst., Vol 257,
No3., March 1954, pp 161-190 and No4., April 1954, pp 275-303

[Hun04] E.V. Huntington,
"Sets of Independent Postulates for the Algebra of Logic",
Trans. Am. Math. Soc., Vol 5, July 1904, pp 288-309

[IEEE88] IEEE Standard VHDL Language Reference Manual,
IEEE Std. 1076-1987,
IEEE, New York, 1988

[Lin80] C.H. Lindsey, S.G. van der Meulen,
Informal introduction to ALGOL 68",
Revised ed., North Holland, Amsterdam, 1980

[Mei86] H. Meijer,
"Programmar: A translator generator",
Ph.D. thesis, University of Nijmegen, 1986

[Mul59] D.E. Muller and W.S. Bartky,
"A Theory of Asynchronous Circuits",
in Proc. Int. Symp. Theory of Switching, Vol 29,
Annals of the Comp. Lab. of Harvard University,
Harvard University Press, 1959, pp 204-243

[New86] I.S. Newton,
"Philosophiae Naturalis Principia Mathematica"
London, 1686

[Ree89] C. van Reeuwijk,
"tm: a code generator for stuctured data interfaces (draft)",
Esprit report E881/b38/TUD/CvR/8905, May 1989.

[Ree90] C. van Reeuwijk (ed.),
"Glass: A system description language and its environment:
Implementation and Maintenance"
Delft, May 1990

[Roo93] L. Rooijakkers,
"The bullet abstractor",
Technical Report, University of Nijmegen, 1993

[Seu90] M. Seutter (ed.),
"Glass: A system description language and its environment:

Introduction and User manuals"
University of Nijmegen, May 1990

eu91] M. Seutter (ed.),
"Glass: A system description language and its environment:
Language Reference manual"
University of Nijmegen, June 1991

'hi93] H. v. Thienen,
"It's about Time"
Ph.D. thesis, University of Nijmegen, 1993

'ur79] D.A. Turner,
Sasl language Manual (rev. ed.)
University of Kent, Kent, England, 1979.

'ur85] D.A.Turner,
"Miranda: A non-strict functional language with polymorphic types",
Functional Programming Languages and Computer Architecture,
Lecture Notes in Computer Science, Vol.201,
Springer-Verlag, Berlin, 1985, pp 1-16.

/os91] E. Voss,
GLAMMAR User Manual,
University of Nijmegen, June 1991

Vat74] D.A. Watt,
"Analysis-oriented two-level grammars",
Ph.D. thesis, University of Glasgow, January 1974

Vij76] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzo
C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker,
Revised Report on the Algorithmic Language ALGOL 68,
Springer-Verlag, Berlin, 1976.

Samenvatting

Hardware-ontwerp is tegenwoordig zo complex geworden dat men eigenlijk niet meer zonder formele technieken of geschikte formalismen kan om deze hardware te beschrijven en te verifiëren. Tijdens het ontwerp moet een ontwerper allerlei aspecten van het systeem in de gaten houden zoals de structuur, het gedrag, de kosten, etc. Al deze aspecten zouden gedekt moeten kunnen worden door het gebruikte beschrijvingsformalisme. De huidige hardwarebeschrijvingstalen scoren op dit gebied onvoldoende: vaak kan men slechts één aspect van de hardware in zo'n taal beschrijven terwijl andere in andere beschrijvingstalen (of zelfs in natuurlijke taal) beschreven moeten worden. Ook kan het voorkomen, zoals in VHDL, dat verschillende beschrijvingen van hetzelfde systeem in één taal worden gegeven om zo verschillende aspecten van deze hardware te beschrijven.

Een andere benadering is die van systeemsemantiek. De grondgedachte hiervan is dat veel fysieke systemen en objecten een grotere variëteit aan modellen benodigen dan niet fysieke, zoals programma's, en derhalve extra aandacht vergen voor notationele zuinigheid in hun beschrijving. Een stuk hardware wordt formeel beschreven middels een zorgvuldige syntax en scope regels gebaseerd op de *gebruikelijk notatie voor functies*. Zo'n beschrijving beschrijft op een formele manier de decompositie van een complex systeem in subsystemen en hoe deze onderling verbonden zijn. *A priori* kennen we geen *interpretatie* (*betekenis*) van deze beschrijving. Men kan echter een interpretatie van zo'n formele beschrijving bekomen door een *semantische functie* toe te passen op deze beschrijving. Een semantische functie beeldt een beschrijving af op een (gewenst) aspect van de hardware, die hierdoor beschreven wordt. Door nu verschillende semantische functies op één beschrijving toe te passen kan men dus een heel scala van aspecten van een systeem vinden.

Het belangrijkste doel van Esprit project Forfun (mei 1986–mei 1990) was een feasibility studie van het principe van systeem semantiek en zo mogelijk een prototype beschrijvingsomgeving te ontwikkelen. In de loop van het project is een beschrijvingstaal met de naam Glass ontwikkeld, waarmee hardware formeel beschreven kan worden. Voor deze taal zijn een aantal programma's en semantische functies ontwikkeld. Deze zijn geïmplementeerd in een aantal klassieke programmeertalen (C, Pascal, Eag, Miranda). We kunnen dit proefschrift daarom ook zien als een engineering rapport van de beschrijvingsomgeving voor Glass.

In dit proefschrift zullen we de taal Funmath gebruiken om semantische functies

te beschrijven. Funmath is een taal waarmee wiskundige objecten beschreven kunnen worden. Men moet hierbij met name denken aan functies over continue en discrete domeinen, welke men vaak tegenkomt in de systeemtheorie, electronica en programmatuur. Funmath heeft echter een veel breder doel zoals het kunnen beschrijven van stukken klassieke wiskunde. Onze geprefereerde bewijsstijl zal die van het transformationeel redeneren zijn, welke ook goed door Funmath ondersteund wordt.

Dit proefschrift is dus gecentreerd rond twee (formele) talen: Glass wordt gebruikt als hardwarebeschrijvingstaal en Funmath om wiskunde en semantische functies mee te bescrijven. In hoofstuk 7 wordt Funmath ook gebruikt om te redeneren over hardware ter verificatie ervan. In diverse hoofdstukken geven we ook aan hoe een semantische functie geschreven in Funmath, kan worden geherformuleerd in een bestaande programmeertaal, zodat deze ook daadwerkelijk op een computer geïmplementeerd kan worden.

In hooofdstuk 1 beschouwen we de basisprincipes van Funmath en de daarbij geprefereerde bewijsstijl van het transformationeel redeneren. We introduceren hier ook een aantal nuttige hoger orde functies en sommige van hun eigenschappen.

In het volgende hoofdstuk onderzoeken we hoe goed klassieke wiskundige gebieden zich laten beschrijven in Funmath. Hierbij gaan we in op drie verschillende wiskundige onderwerpen nl. booleaanse algebra, groepentheorie en topologie. We zullen echter alleen de grondprincipes van deze theorieën beschrijven, aangezien elk in hun volledige detaillering meerdere boeken zouden kunnen vullen. Wat ons speciaal interesseert in dit hoofdstuk is waar de functionele notatie van Funmath beschrijvingen dichter en daardoor makkelijker transformeerbaar maakt dan de klassieke wiskundige notatie.

Het derde hoofdstuk is gewijd aan Glass en het idee van systeem semantiek: de motivering van ons Esprit project. De principes en specifieke syntax van Glass worden hier besproken.

Hoofdstuk 4 beschrijft een aantal simpele semantische functies. Het merendeel hiervan is gebaseerd op een generisch model. Deze semantische functies betreffen het statisch gedrag van digitale circuits volgens een aantal simpele modellen (booleaanse algebra, Eichelberger algebra en tristate logica). Enige kostmodellen worden hier ook gepresenteerd.

Het vijfde hoofdstuk gaat in op de historie van de huidige beschrijvingsomgeving, de problemen die ons werden gesteld bij haar implementatie en sommige van onze oplossingen. Ook wordt hier besproken welke ondersteuning de omgeving levert voor implementatoren van semantische functies. In het bijzonder bespreken we de ondersteuning van de programmeertaal C.

In hoofstuk 6 bespreken we twee semantische functies voor het gedrag voor het gedrag van synchrone sequentiële circuits gebaseerd op functies waarbij de gediscretizeerde tijd als domein wordt genomen. Voor deze semantische functies is wel een uitbreiding van het generisch model uit hoofdstuk 4 nodig. We bespreken ook twee implementaties nl. een in de programmeertaal Miranda, die echter in de praktijk onbruikbaar is en een efficiënte implementatie in de programmeertaal C.

Hoofdstuk 7 brengt een hardware ontwerp methode onder de loep waarmee hardware ontworpen wordt door het bewijzen van zijn eigenschappen. Een ontwerp begint

ɔor uit te gaan van een formele specificatie van zijn hoog niveau abstract gedra
eze wordt dan stap voor stap getransformeerd naar een beschrijving met meer laa
veau karakteristieken en details. Uiteindelijk bekomt men zo een beschrijving d
ʼereenkomt met de eigenschappen van de interconnectie van sommige primitieve con
ɔnenten. Hierop kunnen we nu als het ware de inverse van een semantisch funt
ıepassen, die ons een (correcte) beschrijving in Glass levert van het gespecificeerc
rcuit. We bestuderen dit proces zowel voor een combinationeel als een synchroc
·quentiëel circuit.

Hoofdstuk 8 is veel speculatiever van aard dan de hoofdstukken ervoor. We b
ɔreken er enige gedachten en mogelijke modellen voor semantische functies die h
ıynchrone gedrag van digitale circuits aangaan. Ook gaan we in op twee modellen d
ɔor de auteur in C zijn geïmplementeerd.

In het negende hoofdstuk vergelijken we Glass met een bestaande hardwareb
ːhrijvingstaal nl. VHDL. Hiertoe bespreken we eerst een paar basisideeën van VHD
n te vervolgen met een discussie over de voor– en nadelen van beide beschrijvingstale

Het laatste hoofdstuk is tenslotte gewijd aan wat laatste opmerkingen en co
usies.

Velen hebben aan deze ideeën bijgedragen in allerlei nuttige discussies. De spec
ɜke bijdragen van de auteur in deze zijn:

- Voor Funmath: het onderzoek naar het gebruik van Funmath in klassieke wisku
  de, zoals topologie, en als metataal voor Glass.

- Voor Glass: de taaldefinitie (in het kader van Esprit project 881 – Forfun) en
  implementatie van diverse parsers voor Glass.

- Voor semantische functies: de formele definitie van de meeste en de impleme
  tering van alle semantische functies die in dit proefschrift beschreven worden.

Curriculum Vitae

ın Marc Seutter

29 Sept. '62 Geboren in De Bilt.

Mei '80 Eindexamen VWO B bij de R.K. Scholengemeenschap Canisiu college–Mater Dei.

6 Oct. '83 Kandidaatsexamen Wis– en Natuurkunde met bijvak Sterrenkunc (W1), Katholieke Universiteit Nijmegen.

25 Nov. '83 Propaedeutisch examen Informatica, Katholieke Universiteit Nijm gen.

16 Juni '86 – 31 Juli '87
Junior docent bij de faculteit Wiskunde en Natuurwetenschappe van de universiteit Nijmegen.

26 Juni '87 Doctoraal examen Informatica (cum Laude), Katholieke Universite Nijmegen. Afstudeerrichting: Vertalerbouw.

1 Aug. '87 – 16 Aug. '92
Wetenschappelijk medewerker bij de faculteit Wiskunde en Natuu wetenschappen van de universiteit Nijmegen.

17 Aug. '92 – 16 Sept. '93
Project medewerker bij NWO. (project 612-317-404: A generat based on Extended Affix Grammars for Integrated Environments)

Stellingen

behorende bij het proefschrift

The development of semantic functions for a system description language with multiple interpretations

1. Ondanks zijn bondigheid maakt Funmath het ook mogelijk definities neer te schrijven die voor de meeste mensen onbegrijpelijk zijn.
   Hoofdstuk 1 en 6 van dit proefschrift

2. Het ontwerpen en implementeren van semantische functies is een gesophisticeerde vorm van vertalerbouw.
   Hoofdstuk 4 en 6 van dit proefschrift

3. Echte ervaring in software engineering doet men alleen op in de praktijk.
   Hoofdstuk 5 van dit proefschrift

4. The 42nd corollary of Murphy's Law: Selfcontainment times descriptional power equals a constant.
   Hoofdstuk 1 van dit proefschrift

5. Of het nu om programmeertalen dan wel hardware-beschrijvingstalen gaat schijnt het Amerikaanse Department of Defense met haar voorliefde voor barok eerder in het verleden dan in de toekomst van de informatica te leven.
   Hoofdstuk 9 van dit proefschrift

6. Fortran is een ongeneeslijke infectie van fysici.

7. Veel informaticastudenten beheersen hun wiskunde slecht.

8. Rollenspelen is goed voor je karakterontwikkeling.

9. Onderzoek naar warpveldvergelijkingen leidt automatisch tot een beter begrip van de Algemene Relativiteitstheorie.

10. De regering voldoet in steeds betere mate aan de wet Openbaarheid van Bestuur door het steeds vaker uitlekken van vertrouwelijke rapporten en memo's.

11. Het is een geliefd tijdverdrijf 's zomers van een terrasje het flanerend publiek waar te nemen. Vreemd genoeg wordt deze activiteit vaker met aapjes kijken dan met mensen kijken aangeduid.